Scale-Independent Relational Query Processing

Michael Armbrust

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-165
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-165.html

October 4, 2013

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
04 OCT 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Scale-Independent Relational Query Processing	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California, Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

An increasingly common pattern is for newly-released web applications to succumb to a ?Success Disaster?. In this scenario, overloaded database machines and resultant high response times destroy a previously good user experience, just as a site is becoming popular. Unfortunately, the data independence provided by a traditional relational database system while useful for agile development, only exacerbates the problem by hiding potentially expensive queries under simple declarative expressions. The disconnect between expression complexity and runtime cost often leads developers to mistrust the suitability of relational database systems for their web applications in the long term. As a result, developers of these applications are increasingly abandoning relational systems in favor of imperative code written against distributed ?NoSQL? key/value stores, losing the many benefits of data independence in the process. While some claim that scalability issues are inherent in the use of the relational model this thesis challenges that notion by extending standard data independence with the notion of scale independence. In contrast to traditional relational databases, a scale-independent system is capable of providing predictable response time for all of the queries in an application even as the amount of data grows by orders of magnitude. This predictability is achieved by compile-time enforcement of strict upper bounds on the number of operations that will be performed for all queries. Coupled with a service level objective (SLO) compliance prediction model and a scalable storage architecture, these upper bounds make it easy for developers to write success-tolerant applications that support an arbitrarily large number of users while still providing acceptable performance. Statically bounding the amount of work required to execute a query can be easy for some queries, such as those that perform a lookup of a single record by primary key. However such simple queries are generally insufficient for the construction of complex, real-world applications. Therefore, to enable successful development of such applications, this thesis defines four levels of scale-independent execution that greatly expand the space of queries that can be guaranteed to be scalable a priori. Each scale-independent execution level leverages increasingly sophisticated techniques, ranging from extra cardinality information in the schema to incremental precomputation, to ensure that the performance of the application will not change as the amount of stored data grows. Furthermore, developers can use the levels to reason about the resource requirements of each query that is run by their application. In addition to presenting the theory

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**144**	

Scale-Independent Relational Query Processing

by

Michael Paul Armbrust

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Armando Fox, Chair
Professor Michael J. Franklin
Professor David A. Patterson
Professor Alexandre M. Bayen

Fall 2013

Scale-Independent Relational Query Processing

Michael Paul Armbrust

Abstract

Scale-Independent Relational Query Processing

by

Michael Paul Armbrust

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Armando Fox, Chair

An increasingly common pattern is for newly-released web applications to succumb to a "Success Disaster". In this scenario, overloaded database machines and resultant high response times destroy a previously good user experience, just as a site is becoming popular. Unfortunately, the data independence provided by a traditional relational database system, while useful for agile development, only exacerbates the problem by hiding potentially expensive queries under simple declarative expressions. The disconnect between expression complexity and runtime cost often leads developers to mistrust the suitability of relational database systems for their web applications in the long term. As a result, developers of these applications are increasingly abandoning relational systems in favor of imperative code written against distributed "NoSQL" key/value stores, losing the many benefits of data independence in the process.

While some claim that scalability issues are inherent in the use of the relational model, this thesis challenges that notion by extending standard data independence with the notion of *scale independence*. In contrast to traditional relational databases, a scale-independent system is capable of providing predictable response time for all of the queries in an application, even as the amount of data grows by orders of magnitude. This predictability is achieved by compile-time enforcement of strict upper bounds on the number of operations that will be performed for all queries. Coupled with a service level objective (SLO) compliance prediction model and a scalable storage architecture, these upper bounds make it easy for developers to write *success-tolerant* applications that support an arbitrarily large number of users while still providing acceptable performance.

Statically bounding the amount of work required to execute a query can be easy for some queries, such as those that perform a lookup of a single record by primary key. However, such simple queries are generally insufficient for the construction of complex, real-world applications. Therefore, to enable successful development of such applications, this thesis defines four levels of scale-independent execution that greatly expand the space of queries that can be guaranteed to be scalable a priori. Each scale-independent execution level leverages increasingly sophisticated techniques, ranging from extra cardinality information in

the schema to incremental precomputation, to ensure that the performance of the application will not change as the amount of stored data grows. Furthermore, developers can use the levels to reason about the resource requirements of each query that is run by their application.

In addition to presenting the theory of scale independence, this thesis describes PIQL, a actual implementation of a scalable relational engine. Using the PIQL system, I present an empirical analysis of scale independence that includes all the queries from the TPC-W benchmark and validates PIQL's ability to maintain nearly constant high-quantile query and update latency, even as an application scales to hundreds of machines.

To Henry: Boyfriend, partner, best friend, ghost co-author...

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to the many people without whom this thesis would not have been possible. During my time at Berkeley I was privileged to work with not one, but three amazing advisors. David Patterson and Armando Fox were instrumental in my early graduate career. They taught me how how to find interesting problems and gave me the freedom to explore different topics. It was this freedom that allowed me to discover my interest in scalable data management.

Michael Franklin helped me take my vague ideas about a "performance-safe query language" and turn them into a theory that built upon (instead of outright rejecting) all the previous work on structured data management. I am indebted to him for his insight, patience and guidance.

My experience at Berkeley was greatly enhanced by my interactions with many great collaborators. I was privileged to work closely with two outstanding undergraduates: Stephen Tu and Eric Liang. Their hard work was critical in the process of turning PIQL into a real, working system. Anthony Joseph, Randy Katz, and Joe Hellerstein provided valuable feedback as my ideas on scale independence evolved. Tim Kraska, Beth Trushkowsky, Andy Konwinski, Kristal Curtis, and Peter Bodik were the best labmates collaborators I could have hoped for.

I am also grateful for the many graduate students and post-docs who helped shape not only my research but my whole graduate experience. An incomplete list of these amazing colleagues includes: Ari Rabkin, Charles Sutton, Nick Lanham, Tyson Condie, Peter Alvaro, Neil Conway, Peter Bailis, Jake Abernethy, Rob Carroll, Brian Gawalt, Isabelle Stanton, Sarah Bird, Andrew Waterman, Steve Hanna, Erika Chin, George Schaeffer, Kurt Thomas and Chris Grier.

My interactions with industrial sponsors went far deeper than funding renewals, as they actively helped guide this research to solve real world problems. I am particularly thankful for the insight provided by Bill Bolosky, Kimberly Keeton, James Hamilton, and Pat Helland. Furthermore, this work was supported in part by gifts from Google, SAP, Amazon Web Services, Cloudera, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, IBM, Intel, MarkLogic, Microsoft, NEC Labs, NetApp, Oracle, Samsung, Splunk, VMware, Yahoo!, DARPA (#FA8650-11-C-7136, #FA8750-12-2-0331) and NSF CISE Expeditions (#CCF-1139158).

I thank my family, who provided me with assistance every step of the way. Thanks in particular to my mother, father and brother, whose love, guidance, and encouragement made me who I am today. I am also appreciative of Irene Balassis who, much to my surprise (and delight), actually read my whole thesis and is in no small part responsible for my attempt at grammatical correctness. Finally, it is not possible for me to adequately express how much credit Henry Cook deserves, not only for the successful completion of my graduate work, but for his endless love, companionship, patience and support.

Chapter 1

Introduction

1.1 The Advent of Large-Scale Interactive Applications

The advent of the internet has resulted in substantial changes to the type and scope of possible computer applications. Not only do modern web development frameworks make it easy to build a compelling application in as little time as a weekend, but the connectivity of the internet allows people to experience these new applications simply by typing an address into a web-browser. Together, these changes have resulted in the creation of a new category of fast-growing internet based services, which includes sites such as Amazon, Facebook and Google. Due to their world-wide availability these internet services are often characterized by long periods of extremely rapid growth.

Unfortunately, designing and operating applications that are capable of handling rapidly increasing demand is not trivial. The viral growth rates of these large web applications represent a fundamentally more difficult scalability challenge to developers than pre-internet applications, which only had to provide support for a pool of users that was growing relatively slowly. These sites must contend with a constant influx of new users from around the globe, possibly leading to many years of exponential growth. For example, the popular micro-blogging site Twitter has seen nearly exponential growth in the number of posts made by its users each day [88]. Another popular social networking site, Facebook, recorded over 1.13 trillion "likes[1]" in the first three years that this features was available [49]. That is over 700k likes per minute!

The need to deal with surging popularity is not limited solely to consumer facing social-media applications. With the increasing prevalence of software as a service (SaaS), the developers of important business applications such as customer relationship management (CRM) are also being forced to build software capable of quickly growing to a previously unimaginable scale. Salesforce.com, for example, has gone from servicing 500 million re-

[1] A "like" allows users to tell their friends what content (photos, status updates, products, etc.) they enjoy when using Facebook.

quests/quarter to about 10 billion requests/quarter in four years [65].

Furthermore, the challenge of designing software that scales gracefully with demand is not confined to large companies with huge datacenters. The elasticity provided by cloud computing [8] enables computational resources to be scaled up with only a credit card rather than requiring huge capital investments. Thus, even small developers can have access to a huge number of potential users as well as the computing power to handle them, but only if their application can adapt fast enough.

1.2 Success Disasters

Current software development tools fail to assist developers with the challenges associated with rapid scaling. Thus, many web applications succumb to "success disasters" shortly after becoming popular. In this common scenario, a new web service quickly grows from thousands of users to millions due to a favorable mention on a prominent blog or other news source. Driven by the number of users, the application's database will grow in size by orders of magnitude. This sudden explosion of data often leads to growing pains in the form of unacceptable response times or failed requests.

Research has shown that even a small increase in latency has a measurable effect on user behavior. For example, Google and Microsoft found from user studies that increasing the time it takes for a given page to load by as little as 200 ms resulted in measurable decrease in the number of searches users perform on average [73]. The effect of even these tiny delays was so strong that the researchers were forced to terminate the experiments early for fear of causing actual harm to the popularity of the site.

Decreased user activity is only the first problem that occurs when an application struggles with the growing pains associated with a sudden increase in popularity. The internet has many stories of websites that eventually lost to their competition due to an inability to overcome the scaling hurdle. For example, the downfall of Friendster, a precursor to the wildly successful social-networking site Facebook, is often attributed to its inability to deal with its own early surge in popularity [69].

1.3 Building Applications on a RDBMS

Typically, data management lies at the core of the scalability problem, as most other components of a web application are stateless and thus easily adapt to increased demand. The Relational Database Management System (RDBMS) has long served as a key building block for applications that store and retrieve data, and web applications are no exception. Database systems are used nearly everywhere that computers are found including air travel, banking, retail, defense, scientific computing, and more. These storage systems provide developers with a simple and consistent way of interacting with data while abstracting away challenges

such as efficient storage, retrieval, concurrency, backup, and fault tolerance. These features greatly simplify the task of quickly building performant applications.

Perhaps most importantly, an RDBMS provides developers with a *declarative* query language: SQL. A declarative language allows developers to specify *what* data they want returned without requiring them to say *how* the retrieval will be accomplished. This separation of concerns, known as *data independence*, grants the developer the freedom to focus on the features and user experience of their application without worrying about how changing the representation of the data will affect the performance. Using techniques from over four decades of research, the RDBMS can automatically choose the execution strategy for each query that minimizes the expected cost of computing the answer.

Databases not only make it easier for developers to build the first version of an application, but they can also ease the process of adding features to adapt to changing user needs. This agility is yet another positive benefit developers gain by expressing their application's queries in a declarative language. Since the application does not specify directly how to retrieve the data, developers can add or remove columns from tables or even modify the way data is organized and indexed without the need to change any application code. Instead, it is up to the RDBMS to adapt to these changes by automatically choosing new execution plans for existing queries.

1.4 The NoSQL 'Solution'

Despite the many benefits of their current implementations, database systems are not a panacea. Most strikingly, existing database systems do little to prevent the applications built on them from succumbing to a "success disaster". In fact, the features of modern relational database systems can actually exacerbate the scaling hurdles faced by a suddenly popular service. As proponents of "NoSQL" solutions have widely publicized (e.g., [89, 27]), the declarative, high-level programming interface provided by SQL database systems makes it easy for developers to inadvertently write queries that are prone to scalability problems. The issue is that data independence can hide potentially expensive operations, resulting in queries that perform well over small datasets but fail to meet performance goals as the size of the database grows. Often such performance problems are detected only after they impede site usability, and the database system provides little guidance on how to isolate and fix the problem (assuming a fix is even possible).

To the chagrin of many in the database research community, it is thus increasingly common for web application developers to abandon SQL-style data independence. Instead, these developers choose the straightforward pain of hand-coding imperative queries against distributed key/value stores. This decision is motivated by the fact that many key/value stores are capable of providing linear scalability as machines are added and of maintaining predictable per-request latencies [31]. However, this approach creates its own problems. The use of imperative functions instead of declarative queries means that changes to the data model often require time-consuming rewrites of application code. Perhaps more critically,

developers are forced to manually parallelize requests to avoid the delays of sequential execution. Therefore, the benefits of physical and logical data independence are lost.

1.5 Introducing Scale Independence

The status quo of data management leaves today's developers with two unacceptable alternatives. Cost-conscious organizations may choose to ignore the possibility of future scalability issues, opting instead to focus their efforts on adding features to their application. Past experience has shown that such sites often end up as victims of their own success.

In contrast, forward-thinking developers may decide to over-engineer their systems so as to meet arbitrarily high scalability targets. Unfortunately, this strategy is equally untenable in practice. Designing for massive scale can squander precious resources during the critical early stages of application development with no assurance that the application will ever actually become popular. Had these developers been free instead to focus their efforts on feature development, perhaps their application would have been more successful in attracting users. Even in the cases where the costs of this over-engineering are eventually warranted, any chosen scalability target may still end up being overly conservative.

Ideally, developers should be given the tools to develop data-intensive applications rapidly without needing to worry about meeting some fixed scalability target. Such tools would ensure that applications will perform predictably as they grow in popularity, while simultaneously preserving the many productivity benefits of the relational model. As I demonstrate in this thesis, this syncretism can be accomplished through the introduction of *scale independence* [9], a new type of data independence. Scale independence provides a new way to reason about building data management systems, and my thesis presents the theory, algorithms, and tools that enable this approach for a wide range of relational queries. Scale-independent queries that satisfy their performance objectives on small data sizes will continue to meet those objectives as the database size grows, even in a hyper-growth situation such as when a web service goes viral. A scale-independent system is inherently *success-tolerant*, making it easy for developers to ensure that their initial implementation will be able to handle the massive onslaught of data that is characteristic of success on the web.

1.6 Implementation of a Scale-Independent Relational System

The mechanism used in this thesis for achieving scale independence is the calculation and enforcement of an upper bound on the number of storage operations that a query will perform *regardless of the size of the underlying database*. This strategy can be effective at ensuring predictable performance during period of rapid growth, as the number of storage operations performed is often the dominant driver of latency in query execution. However, simply mandating the existence of an upper bound for every query does not accomplish

the goal of helping developers build success-tolerant applications for two reasons. First, for interactive applications, performance objectives are typically based on response time rather than operation count. Therefore, queries whose execution requires an unreasonably large (though bounded) number of operations could still fail to perform acceptably. Second, naïvely adapting existing relational database systems to simply reject queries where no bounded plan exists results in a large number of false negatives, excluding queries that could easily be made scale-independent. Instead of simply restricting the space of possible execution plans, a scale-independent storage system should help developers by suggesting possible modifications to queries and the application's schema.

A scale independent relational system uses static analysis to only allow query plans in which it can calculate a bound on the number of key/value operations to be performed at every step in their execution. Therefore, in contrast to traditional query optimizers, the objective function of the query compiler is not to find the plan that is fastest on average. Rather, the goal is to avoid performance degradation as the database grows. Thus, the compiler will choose a potentially slower bounded plan over an unbounded plan that happens to be faster given the current database statistics.

While other systems, such as GQL [37] and CQL [20], also provide a SQL-like query language that bounds computation, they impose severe functional restrictions, such as removing joins, in order to ensure scalability. In contrast, a scale-independent system will also avoid inherently unscalable queries, but it employs language extensions, query compilation technology, precomputation, and response-time estimation to provide scale independence over a larger and more powerful subset of SQL.

If the query compiler cannot create a bounded plan for a query, it warns the developer and suggests possible ways to bound the computation. This static analysis can be performed for some queries using existing annotations, such as the `LIMIT` clause [19] or foreign key constraints. However, in many cases, it is insufficient to simply limit the result size, as intermediate steps also contribute to execution time. Therefore, in this thesis I extend SQL to allow developers to provide extra bounding information to the compiler. First, my extended SQL syntax provides a `PAGINATE` clause, allowing the results of unbounded queries to be efficiently traversed, one scale-independent interaction at a time. Second, I provide constructs that enable bounding intermediate results through relationship cardinality constraints expressed in the database schema. Together, these additions allow developers to express complex queries while still providing the query optimizer with enough information to determine that these queries will not result in performance problems as the application grows.

To avoid choosing plans that perform too many storage operations, I show that it is possible to employ a worst case performance prediction model. Specifically, I describe techniques that allow developers to specify Service Level Objectives (SLOs), which are framed as a target response time for a fraction of the queries observed during a given time interval; e.g., "99% of queries during each ten-minute interval should complete in under 500 ms." Given this information, my SLO compliance prediction model uses the query plan and the operation bounds to calculate the likelihood of a scale-independent query executing with

acceptable performance. Section 4.4 demonstrates that, for a variety of benchmark queries, even a simple model can accurately predict SLO compliance. Intuitively, such prediction is possible due to a combination of the predictability of the underlying storage system and the bounds on the number of operations each PIQL query will perform in the worst case. It is important to note that the focus of this thesis is on interactive queries, as these often have the most stringent performance requirements. These queries generally require relatively few accesses and there is typically sufficient headroom between these conservative bounds and the response time requirements of the application.

Finally, there are SQL queries, common in real world applications, where it is simply impossible to bound the number of storage operations required if all processing is performed *on-demand* when the answer is desired. For example, the online service Twitter needs to calculate the number of people following popular users. Executing this query on-demand could result in response time that grows with the size of the database. Fortunately, it is often possible to answer such queries safely at scale by leveraging *incremental precomputation*, effectively shifting some query processing work from execution time to insertion time.

I formally define the classes of SQL queries where precomputation fundamentally changes the worst case execution cost at scale. Understanding the characteristics of these classes allows the construction of a scale-independent view selection and maintenance system. My scale independent view selection system [10] is unlike prior work on materialized views [3, 51, 60], which attempted to minimize cost of query execution for a given workload. While these prior techniques select views that may speed up the time it takes to answer a query on average, the overall performance of the application can unfortunately remain dependent on the size of the underlying database, thereby violating scale independence. Instead, a scale independent system focuses on ensuring predictable performance, even as the data size and workload grow.

1.6.1 Impact of Scale-Independent Thinking

Requiring the presence of a strict bound on the number of operations that will be performed in the worst case means that a scale-independent optimizer will prioritize scale-independent plans over those that are cost-optimal on average. As a result, there are cases where such a system will select a plan that does not execute as quickly as possible over small amounts of data. While this choice may appear counter-intuitive, it is made by the optimizer in the hopes that the bounded plan will exhibit more predictable behaviour as the amount of data in the system grows.

In order to quantify the effect of scale-independent plan selection on response time, one can study what happens when a query known to cause scalability problems is executed over different amounts of data using both a scale-independent plan and one that is cost-optimal on average. The query is used by the microblogging site Twitter and checks to see which of the current user's friends are also subscribed to the user whose profile is being viewed. Figure 1.1 shows where the results of this query are rendered in the Twitter interface.

The schema and SQL for this query are as follows:

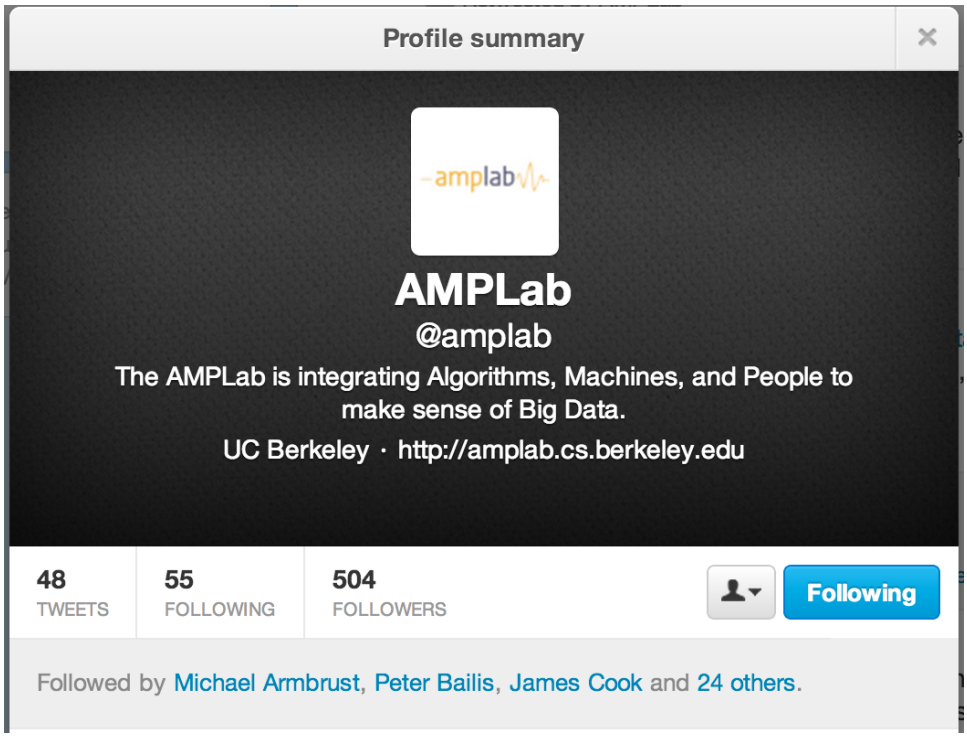

Figure 1.1: Twitter shows a list of common subscribers when viewing the profile of another user.

```
Subscriptions(ownerId, targetId, approved) WITH CARDINALITY(ownerId, 50)

SELECT * FROM Subscriptions
WHERE targetId = <target user>
AND ownerId IN <users followed by the current user>
```

For the purpose of this experiment I create two indexes over the Subscriptions relation. The first is the clustered index over the primary key (`ownerId`, `targetId`). The other is a secondary index over (`targetId`).

Given the above physical schema, a query optimizer must decide between two different physical plans for executing the above query. The first option performs an unbounded sequential scan over the secondary index to retrieve the list of all subscriptions that match the predicate `targetId = <target user>`. The results of this scan can then be compared locally with the list of users followed by the current user. To determine the I/O cost of this plan, the optimizer will look at statistics on the average number of followers for any given user on Twitter. Since this number is small — 126 in 2009 [11] — and the results will be contiguous, the optimizer determines the average cost of this plan to be a single I/O operation.

The second possible physical plan computes the answer to the query using the clustered primary index. In this case, the query is executed by probing the clustered index, checking for the presence of a record for each id from the list of users followed by the current user. Since these tuples are not necessary contiguous, each probe will require a random I/O. However, due to the cardinality constraint on the number of subscriptions for any given owner, the optimizer knows that no more than 50 I/O operations will be required in the worst case.

Presented with these two options, a cost-based optimizer will clearly choose the first plan. In contrast, a scale-independent optimizer will not even consider this plan, due to its potentially unbounded I/O costs. To understand empirically how these decisions will affect overall response time, each plan is hand-coded and then run against users of increasing popularity. For each data point, 50 randomly-selected users are selected as the friends of the current user. Figure 1.2 shows how the response time changes with the popularity of the target user.

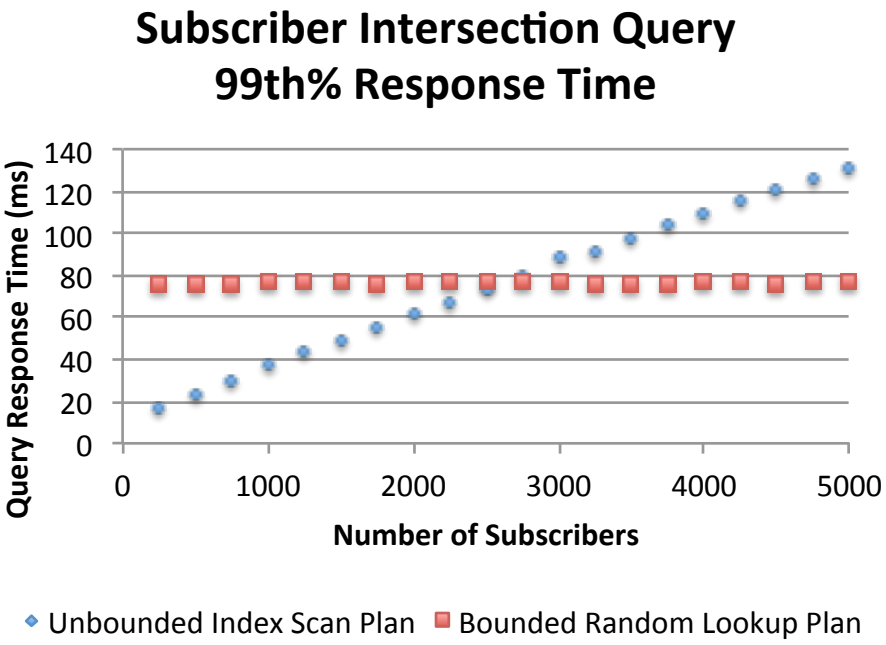

Figure 1.2: A comparison of the 99th percentile response time of 200,000 executions of the subscriber intersection query using two different optimization strategies.

While the cost-based plan performs up to 4x faster for an unpopular user, the scale-independent plan consistently meets the application's SLO, independent of the popularity of the target user. For a popular user, such as "Lady GaGa" (12M+ followers), using the cost-based query plan would certainly violate the SLO.

This experiment empirically demonstrates two key facts. First, the cost of I/O operations against the key/value store is often the dominant driver of latency in query execution.

Therefore, predictable performance can often be obtained by bounding the number of storage operations that will be performed in the worst case. Second, while cost-based optimization can reduce average query response time, it is not sufficient to ensure SLO compliance during periods of rapid growth.

Additionally, if it is desirable to further minimize response time, even below the SLO, a scale-independent system could be extended to use a dynamic approach that determines at runtime which query plan to execute. However, special care would need to be taken to ensure that unbounded plans were never run over unsafe amounts of data.

1.7 Summary and Contributions

In this thesis, I demonstrate that it is possible to provide the developers of interactive applications with both the productivity enhancing features of the RDBMS and the ability to endure periods of extremely rapid growth. I begin by first formally defining the characteristics of scalable queries. Then, I use this formalism to construct different classes of SQL queries, segmenting them based on the techniques that are required to ensure that they scale gracefully.

To demonstrate the effectiveness of this approach, I also present PIQL[2], a first attempt at building a scale-independent RDBMS, which is capable of running on hundreds of machines. The PIQL system builds on decades of database system research, but shifts the focus from raw performance to long-term scalability. PIQL is capable of both determining the inherent scalability of all the queries in an application and ensuring they execute with predictable performance as both data and the number of machines grow. This feat is accomplished by rethinking the implementation at all layers, including the query language, optimizer, view selection system, as well as the underlying storage system. In summary, this thesis contains the following contributions:

- I create and define the notion of a scale-independent storage system, which ensures that applications will perform predictably even during hyper-growth situations.
- I formalize the invariants on computational resources that must be maintained by a scale-independent system.
- Using these invariants, I define four levels of scale-independent query execution based on the resources required during execution.
- I present a minimal extension to SQL that allows developers to express relationship cardinality and result size requirements.
- I describe a scale independent query compiler, which bounds the number of key/value store operations performed for a given query.

[2]Short for the Performance Insightful Query Language and pronounced “pickle”.

- I present a performance model that helps developers determine acceptable relationship cardinalities and reason about SLO compliance.
- I present a scale-independent view construction algorithm along with static analysis techniques to bound the cost of storage and maintenance.
- I describe a mechanism for automatically detecting and mitigating common temporal hotspots using a combination of load balancing and parallel execution.
- I present the results of empirical studies that demonstrates the expressiveness of the extended SQL language, the accuracy of the SLO compliance prediction, and the scale independence of the PIQL implementation using two benchmarks running on hundreds of machines.

1.8 Organization

The rest of this thesis is organized as follows: Chapter 2 contains the necessary background for understanding scale independence. Section 2.2 describes the relational model, which serves as the theoretical underpinning of the modern RDBMS and the basis for data independence. Readers familiar with relational algebra and standard database architecture can safely skip much of this section. However, Section 2.3 contains specifics on materialized views selection and maintenance, which are less commonly known. Next, Section 2.4 explains the characteristics and architecture of large scale web services, which serve as a motivating example of the importance of scale independence. Finally, Section 2.5 describes key/value stores, which serve as a common alternative to relational database systems for scale oriented developers.

Chapter 3 formally introduces the notion of scale independence. It begins by explaining the notion of "scale" as well as how the amount of data returned by various queries will change as a system grows. Next, it lists the invariants on resource consumption that serve as the basis for for a scale-independent system. Using these invariants, this chapter then lays out four levels of scale-independent query execution based on the secondary structures required for their execution. By understanding which level will be used to execute a given query, developers can better understand the resource consumption that will be incurred while executing the query as the system grows. Finally, this chapter describes how the enforcement of these invariants can map to query response time and thus user experience in a correctly designed system.

Given that the invariants maintained by a scale independent system have been enumerated, Chapter 4 examines how a database can obey these invariants while executing relational queries on demand. First, the algorithm that determines if a query has a scale-independent execution plan is presented. The chapter goes on to describe how empirical measurements of key/value store performance can be used to predict whether new queries will meet response time goals as an application grows.

Chapter 5 expands on Chapter 4 by addressing queries where precomputation is required for scale-independent execution. Since materialized views must now be created, this chapter also explains the static analysis required to ensure that the cost of storing and maintaining these views will not itself threaten scalability.

The implementation of PIQL would not be possible without a scalable underlying storage system. Chapter 6 describes the SCADS [9] key/value store used for my empirical evaluations, focusing on the architectural decisions that were important for achieving scale independence.

Finally, Chapter 7 discusses the limitations of the current implementation and lists future directions for research on scale-independent query processing. It concludes with a summary of the contributions of this work.

Chapter 2

Background

2.1 Introduction

In order to understand scale independence, it is important initially to look at current storage systems and how they help developers rapidly build complex applications. This chapter begins with a description of the relational model and discusses how this model enables data independence. Sections 2.2.1-2.2.3 give an overview of relational algebra as well as the architecture used by a modern RDBMS to execute a query expressed relationally. This material is fairly basic in nature and can safely be skipped by readers familiar with relational query execution. Building on this foundation, Section 2.3 discusses the selection and incremental maintenance of materialized views, focusing on techniques that are extended by PIQL when precomputing the results of queries when required for scale independence.

Section 2.4 introduces one motivating use case for a scale-independent storage system: Large-scale interactive web services. This section starts by discussing the characteristics of these applications, contrasting them with more traditional pieces of software. Next, it discusses how these applications attempt to adapt to a rapidly growing user base and the pain points they commonly experience during this process.

Finally, Section 2.5 concludes the background material by describing NoSQL systems, which serve both as a feature-anemic alternative to an RDBMS as well as the underlying storage system for my implementation of a scale-independent system, PIQL.

2.2 Review of Relational Database Technology[1]

While the first computers were designed for "pure" computing tasks like calculating missile trajectories or breaking complex encryption codes, it was quickly realized they were also useful for storing and retrieving information. As a result, a significant amount of research

[1]This chaper is intended to serve as a primer on the necessary background material for readers who are less acquainted with prior work on relational query processing. Those readers who are already familiar with the relational model and the architecture of the traditional RDBMs can safely skip this section.

and engineering effort has gone into making it easier for developers of computer applications to manage data.

The earliest data management systems were built directly on top of file systems. While file systems assist developers by abstracting away the challenge of manually interfacing with storage devices, they fall short in many regards. Most importantly, file systems provide no support for quickly accessing data unless the location within a given file is already known. This omission means that developers concerned about performance need to worry about maintaining structures, such as hash tables or binary trees, that allow faster access to data based on content.

The inefficiency of forcing each developer to reinvent abstractions from scratch for each data management challenge encountered quickly led to the creation of more generalized data storage systems. This trend towards generalization began with navigational databases in the 1960s and became standardized by the CODASYL group [25].

Navigational databases presented developers with a graph abstraction where each data item contained pointers to other data items. Upon opening a database, the developer would receive a pointer to the "first" record. Queries were written by writing programs that would "navigate" through this graph by following pointers to retrieve the desired data.

This data access paradigm represented a significant advance over simple file systems, as developers no longer needed to maintain the storage structures that allow a program to look up records efficiently given a pointer. However, due to the lack of a built-in support more complex that following pointers, several issues remained. First, answering a query by navigating over the provided graph data structure still required the developer to write a significant amount of code. Second, because query code was closely coupled with the structure of the graph, any changes to the way data is stored could require significant modifications to application code.

2.2.1 The Relational Model

Codd addressed the challenges presented by closely coupled application code and physical storage of data through the introduction of the relational model, as reported in his seminal paper "A Relational Model of Data for Large Shared Data Banks" [26]. In this paper, Codd describes a general model for both representing and querying data stored by a computer.

At the highest level, the relational model represents data as a set of *relations* (also known as *tables*). Each relation is comprised of a set of unordered *tuples* (also called *rows*). A tuple is an ordered set of *attributes* (also known as *columns*) that can be optionally identified by a unique key. The *schema*, or the names of the attributes and their data types (i.e., String, Integer, Boolean, etc), is the same for all tuples in a relation.

To better understand this model, consider a simple system that stores a list of books along with the authors who wrote them. Figure 2.1 shows the parts of the relational data model as they apply to this schema, along with some sample data. Each book is represented by a tuple in the Books relation (or relation instance). The author for a given book can be found by looking up the tuple in Authors whose ID matches the AuthorID from the

appropriate book tuple.

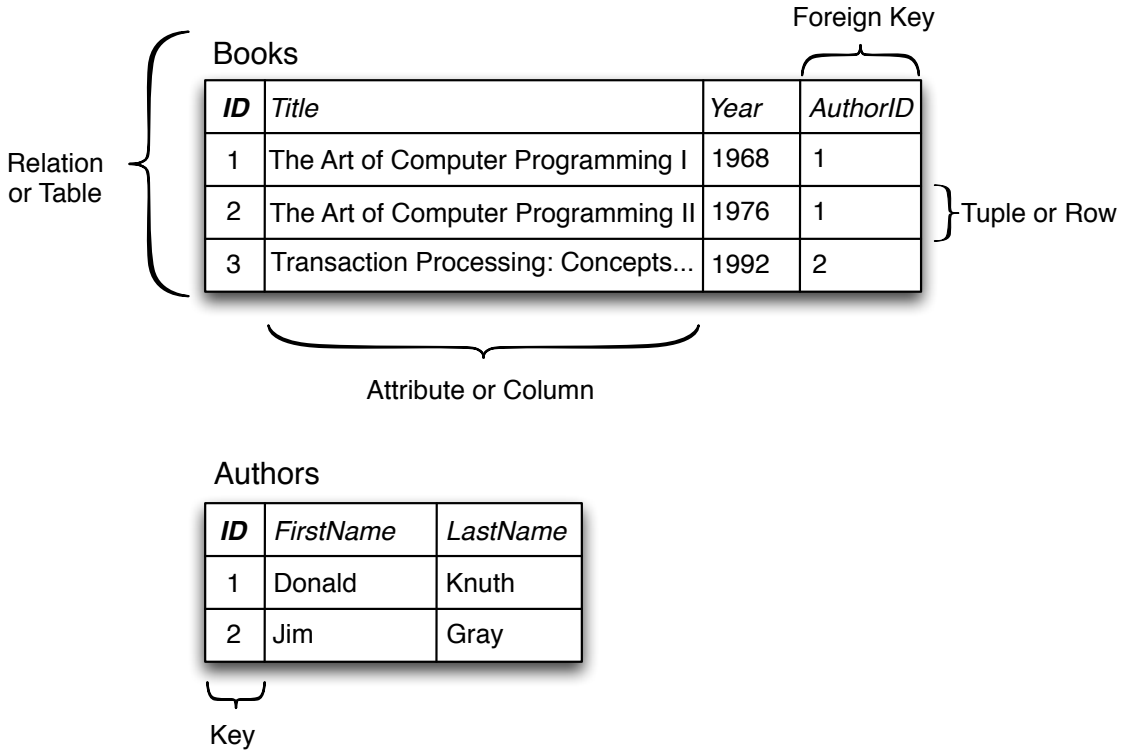

Figure 2.1: An example of a relational schema for storing books and their authors.

While this model provides a very general scheme for representing many different types of data stored in a computer, simply storing the data is insufficient. The real power of the relational model comes from the simple, composable primitives it defines for retrieving data. These data manipulation primitives are built on a mathematical concept known as *relational algebra*[2] and together represent a general model of transforming and querying data.

Relational algebra defines simple operations, such as *union* and *project*, that transform relations, producing new relations. While each of the operations in this algebra only perform a simple action on a relation, by combining them it is possible to express a wide range of queries. Further work done by Hall and others revised this model to include a rename operator [43]. For simplicity of exposition, the following list describes a representative subset of relational operators that are sufficient to express all the queries relevant to applications targeted by this thesis.

[2]Codd was also responsible for the definition of relational calculus, which was later shown to be equivalent to relational algebra.

Set Operators Relational algebra supports four generic set operations that can performed on relations with identical schemas. The schemas of the two relations must be identical because otherwise the result could be a set containing tuples with differing schemas and, therefore, would not be a relation. These operations are described as they apply to two arbitrary sets, R and S.

Union $R \cup S$, the union of R and S, is a set containing all tuples found in *either* R or S. Duplicate items that appear in both sets will only appear in the result set once.

Intersection $R \cap S$, the intersection of R and S, is a set that contains all tuples that appear in *both* R and S.

Difference $R - S$, the difference of R and S, is the set containing all elements of R that do not appear in S. Unlike the two previous operations, the difference operator is not *commutative*, meaning $R - S$ is not necessarily equal to $S - R$.

Cartesian Product

The Cartesian product operator, $\times$, takes two relations and returns each tuple from the first relation matched with all tuples from the second relation. The schemata for the resulting relations is the union of the schema of the two original relations.

For example, using the data from Figure 2.1 once again, the expression $Books \times Authors$ produces the following result:

title	*year*	*authorID*	*ID*	*FirstName*	*LastName*
The Art . . . I	1968	1	1	Donald	Knuth
The Art . . . I	1968	1	2	Jim	Gray
The Art . . . II	1976	1	1	Donald	Knuth
The Art . . . II	1976	1	2	Jim	Gray
Transaction Processing. . .	1992	2	1	Donald	Knuth
Transaction Processing. . .	1992	2	2	Jim	Gray

Other Relational Operators In addition to the standard set operations, relational algebra defines:

Project

The projection operator, π, takes a relation R and produces a copy of that relation with only a subset of the attributes present in R.

For example, using the data from Figure 2.1, the expression $\pi_{title}(Books)$ would produce the following result:

title
The Art of Computer Programming I
The Art of Computer Programming II
Transaction Processing: Concepts and Techniques

Select

The selection operator, σ, can be applied to a relation R and will produce a subset of the tuples present in R. Tuples are included in the emitted subset if and only if they match the specified predicate.

For example, using the data from Figure 2.1, the relational expression $\sigma_{year=1968}(Books)$ returns only the tuples where the year attribute is equal to the value "1968" and produces the following result:

title	*year*	*authorID*
The Art of Computer Programming I	1968	1

By combining these simple operations it is possible to express complex queries. For example, consider the query "Return the last name of all authors who wrote a book during the year 1976". This query can be answered by evaluating the following relational algebra expression.

$$\pi_{LastName}(\sigma_{year='1976'}(\sigma_{AuthorID=ID}(Books \times Authors))) \tag{2.1}$$

An additional feature of defining an algebra for expressing queries is that this formalism provides a set of rules that specify how queries can be transformed without changing what data is returned. This flexibility often allows a database system to execute a query more efficiently than the way it was originally expressed. As a concrete example, consider the query defined in 2.1. As the query is expressed, execution would begin by computing the Cartesian product of the *Books* and *Authors* relations, a potentially expensive operation. However, the same result can be obtained if the *Books* relations is first filtered to remove books that were not written in 1976. The more efficient equivalent query is expressed as follows:

$$\pi_{LastName}(\sigma_{AuthorID=ID}(\sigma_{year='1976'}(Books) \times Authors))) \tag{2.2}$$

This type of optimization is known as *predicate push-down* as it involves "pushing" the predicate deeper in the plan, where it can hopefully filter tuples earlier and reduce the total amount of work required to answer the query. Predicate push-down is only one of many transformations utilized by database systems to improve the efficiency of query execution.

Another important optimization involves the introduction of the *join* operator. Logically, a join is a Cartesian product followed by a selection that only returns tuples where the appropriate attributes match. However, for efficiency reasons, the full results of the Cartesian product are never actually produced when performing this computation. Section 2.2.3.2 describes different types of joins, as well as the analysis performed by the database system to determine which version of a query is the "best".

For practical reasons, actual implementations of the relational model differ from the mathematical formalism described above in several ways. First, pure relational algebra

operates on sets of tuples, meaning that there can be no duplicate tuples in a relation. Real-word implementations often forgo this restriction both for greater expressiveness and performance. However, they usually provide a mechanism for removing duplicates from a result set (e.g. the `DISTINCT` clause in SQL).

Additionally, while sets are generally considered to be unordered collections, it is often useful in practice for the database to support sorting tuples. The utility of enabling the database system to understand sort-order is two-fold. First, it is common to want only a fixed number of results for a given query. For example, consider a query that returns the ten most popular books. Such a query can be executed more efficiently if the database engine understands ordering and thus can return more popular books first. Section 2.2.4 describes the mechanisms for enabling these partial-result queries (often known as *top-K queries*) in greater detail. Second, when the database is aware of the ordering for a set of tuples, either due to an explicit sort operation or due to ordering properties of the data structure the tuples are retrieved from, it can choose more efficient methods of performing logical operations (e.g. the database system can leverage the fact that duplicate tuples will be contiguous in a sorted relation).

A final common addition to the relational model is support for aggregation. Aggregate operators combine multiple tuples, producing a single result tuple. A simple example is an operator that simply counts the number of input tuples, returning the final count. Other common aggregation functions include sum, average, and standard deviation.

In addition to computing aggregate values over all of the tuples in a given result, it is also possible to produce sub-aggregates for *groups* of tuples that share a common value for specified attributes. For example, by grouping on the AuthorID attribute, it would be possible to produce counts of books for each author in the database.

2.2.2 Data Independence

Expressing data manipulation as a set of relational algebra operations was a huge step forward as it allowed developers to express their intent *declaratively*, instead of requiring them to specify exactly how the data will be retrieved. In contrast, queries expressed using earlier data management tools were often dependent on the specific representation of the data. Codd's original paper enumerated three types of this *data dependence*: Ordering dependence, indexing dependence, and access path dependence. Such dependence on the specific layout of data often makes it difficult to change characteristics of data representation without changing the semantics of queries that access the data.

In contrast to programs that directly access data storage primitives to retrieve data, queries expressed as relational algebra exhibit a useful property known as *data independence*. Data independence can be a boon, especially for developers of rapidly changing applications, as it allows them to make significant modifications to the representation of stored data without needing to make any changes to application code. Typically, the degrees of representational freedom provided by data independence are broken up into two categories: Logical and physical.

Logical Independence Logical data independence refers to the flexibility to change the schema definition for a given collection of data. This form of independence means that, for example, attributes can be added to a relation without requiring changes to queries. Additionally, this independence implies that the correctness of queries that operate on a subset of the attributes in a relation will not be affected if attributes outside of this subset are removed.

Logical data independence is also enabled through the concept of *views*. A view is a virtual relation that is specified by a relational expression. By creating a view, a developer can make significant changes to the schema of an application, including modifications such as changing foreign key relationships, and yet still provide an equivalent final data representation to the application. Views can be purely logical, meaning they are evaluated at run-time. However, a potentially more performant option is to *materialize*, or precompute and store the view. Section 2.3 discusses trade-offs associated with the creation and maintenance of materialized views in greater detail.

Physical Independence Physical independence refers to the ability to change the physical representation of the data as it is stored in the memory or on the disk of the computer. Possible changes to the physical schema include: Changing the ordering, storage structure, indexes, file layout, or storage device. Relational database systems not only shield developers from worrying about these types of modifications, but typically also re-optimize queries in an attempt to find the best execution strategy given the new physical layout.

Executing Relational Algebra Expressing queries declaratively using relational algebra means that at some point the RDBMS must convert the query expression into a set of operations that realize the query. The specific set of operations that are used to evaluate a given relational algebra expression are known as a *query plan*. While the rules of relational algebra describe the types of transformations that can occur without changing the semantics of the query, they do not help in finding the best method amongst the many alternatives. Instead, this selection is performed using a process known as *query optimization*. Section 2.2.3 provides the background on how modern systems perform this optimization.

While relational database systems generally do an acceptable job of selecting plans that minimize average response time, they do not help developers reason about how the performance of an application will change as the amount of data grows. In fact, data independence can often make such reasoning significantly more difficult. Specifically, the fact that query plans can change with the specific characteristics of the data stored means that data growth can result in the reoptimization of a previously well-performing query. In contrast, my system, PIQL, provides standard data independence but extends it with scale independence.

In the next section, the execution of a relational query is explained in the context of the architecture of a modern relational database system. Special care is taken to highlight the parts of the system that need to be adapted to provide scale independence.

2.2.3 Architecture

Many existing data management systems are capable of executing relational queries, including commercial systems like Microsoft SQLServer, Oracle, and IBM DB2. Several open source options are also available, including Postgresql, MySQL, and SQLite. These modern relational databases are generally very complex software systems. Even PostgreSQL, a relatively simple RDBMS compared to its more expensive commercial counterparts, is comprised of over 600,000 lines of code [79]. As such, architecturally an RDBMS is generally broken down into several layers, allowing the higher layers to build on top of the abstractions provided by the lower layers. Figure 2.2 shows how the layers of an RDBMS take a query written in SQL and eventually translate it to operations that retrieve the requested data. More importantly, since most systems share this common architecture, advances made by one particular implementation can often be easily applied to others. This section explains the functionality of each of the layers of an RDBMS, highlighting the components that need to be modified to provide scale-independence. For the sake of brevity, I omit details of subsystems that primarily effect the relative performance of query execution and not scalability (e.g. logging and recovery).

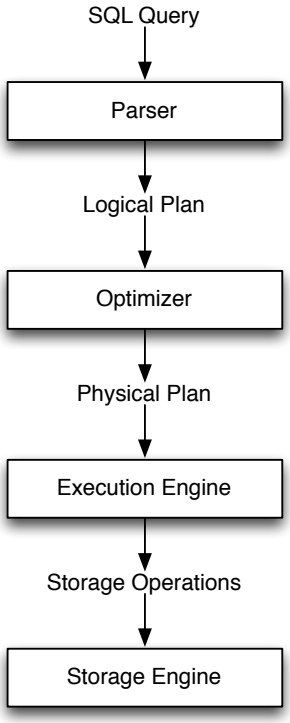

Figure 2.2: The phases of query execution through the architecture of a standard relational database system.

2.2.3.1 Parser

The first phase of SQL query execution occurs in the parser. The parser takes in a SQL statement in the form of a string and generates a parse tree of the expression. This parse tree is then converted into a logical query plan composed of nodes that closely map to the relational algebra expressions from the previous section. The logical query plan serves as a representation of what data will be returned by the query, but does not yet specify details such as which indexes will be used or which algorithms will be invoked for complex operations such as joins.

As an example, consider the query from the previous section, which returns the last name of all authors who wrote a book in 1976. A developer would express this query to the database with the following SQL:

```
SELECT DISTINCT LastName
FROM Authors, Books
WHERE AuthorID = ID AND
      Year = 1976
```

Figure 2.3 shows the logical query plan that is produced by the parser given the previous SQL expression. The query plan should be bottom up starting with the relations at the bottom. The final results are produced by the top operator.

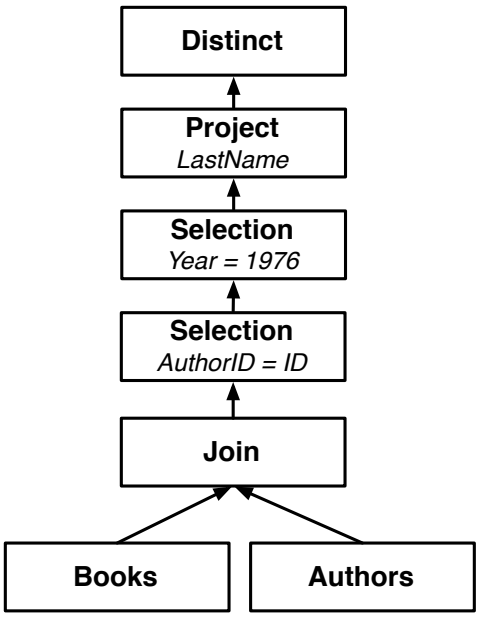

Figure 2.3: The logical plan for a query that returns the last name of authors who wrote a book in 1976.

2.2.3.2 Optimizer

The optimizer serves as a bridge between the programmer's expression of what query is to be run and the execution engine that actually answers the query. It starts with the logical query plan created by the parser and creates a physical query plan that is expected to be most efficient.

In the first phase, the optimizer performs algebraic rewrites of the query. These algebraic transformations produce semantically equivalent queries that are expected to execute more quickly than the original query. One example of such a transformation was discussed earlier and is known as "predicate push-down". This transformation involves pushing selection operators down deeper into the query plan. Pushing these filters deeper into the query plan almost always results in faster query execution since removing tuples that do not match a predicate early prevents wasted work later in the query plan.

Another example of a common algebraic rewrite is the push-down of *stop operators*, which enable the efficient calculation of partial results. Section 2.2.4.1 discusses stop operators and the rules for pushing them down query plans in greater detail.

After purely algebraic transformations, the optimizer must next generate a physical query plan. In doing so, it must choose the specific access methods and algorithms that will be used to execute each operator of the query.

One part of this process is *access path selection*, which chooses the method that will be used to retrieve the data requested by the query. The simplest access path, though not necessarily the most efficient, is a table scan which simply reads all tuples in a given relation in an unspecified order. While this approach will always produce the correct answer eventually, more efficient methods also exist. For example, often the relation is stored in a specific order on disk (known as a *clustered index*) and this ordering can be used to locate data quickly that satisfies a predicate or even efficiently retrieve data as required by an `ORDER BY` clause.

Even when the ordering of the relation is not helpful for answering the query, other efficient access methods exist. For example, database systems also support the ability to create different types of *secondary indexes*, which can be used to retrieve data quickly based on the value of attributes other than those that define the sorting of the clustered index. The two types of indexes that are most commonly supported are *B-tree* and hash table.

A B-tree is a data structure similar to a binary tree and stores all of the records in sorted order. However, instead of each node of the tree having only two children, each B-tree node has many. This allows for much larger nodes, each of which can be retrieved by a single disk operation. Since B-trees store data in order, lookups over the sorting attributes can be performed efficiently using using binary search. Furthermore, B-trees can also be used to execute range queries (e.g. return all records with dates between January 1st and January 31st). Finally, the ordering property of this type of index can be used to satisfy an `ORDER BY` or to enable the use of an algorithm that relies on tuple ordering later in the plan.

A hash table, on the other hand, stores each data item in a location that is determined by a mathematical hash of the attributes specified. Since hashing these values does not lead to

any useful ordering, hash indexes, in contrast to B-trees, can only be used to locate records by exact equality of the attributes indexed.

In addition to choosing the access paths for each relation in the query, the optimizer also needs to choose which algorithms will be used to satisfy other operations in the plan. For example, the optimizer needs to choose which algorithm to use for each join found in the logical query plan. While in the previous section on relational algebra joins were described as a Cartesian product followed by a selection, this execution strategy is virtually never used in practice due its inefficiency. Instead, a specific join algorithm is chosen based on properties such as the ordering of the data, the size of each relation, and the predicates to be applied to the joined tuples.

One possible join algorithm that is used in practice is known as the *nested loop join*. This algorithm starts by reading tuples one by one from the first relation (known as the *outer relation*). Next, this algorithm scans each tuple in the other relation (known as the *inner relation*) comparing the attributes being joined on. If the attributes match, then a tuple is produced, otherwise the algorithm proceeds to the next tuple. This algorithm is particularly nice because it does not require the creation of any secondary structures and allows tuples to processed one at a time. The benefits of processing tuples one a time are explained in the next section during a discussion of the iterator model.

For a nested loop join, the number of comparisons that need to be performed is $O(mn)$ where m is the size of the outer relation and n is the size of the inner relation. Thus, this join algorithm is often chosen when the size of one of the relations is small.

In cases where the inner relation is actually large, a variation known as an index-loop-join can be used instead. In this algorithm, the inner loop of the nested join is replaced with an index lookup. Since each index lookup can be performed in $O(1)$ or $O(\log n)$, given a hash index or B-tree index respectively, this can result in significantly faster execution at the expense of maintaining an extra index.

Other join algorithms exist such as *hash-join*, *mergesort-join*, as well as many others. These are omitted from this section because they all require reading all the data of a relation before beginning the join operation and thus are inherently not scale independent.

Given all of these choices, the database system must decide which physical plan to actually use for query execution. In most systems, this decision is made using a technique known as *cost-based optimization*. In the simplest case, the notion of cost used by the database system is the number of I/O operations that it expects the query to perform based on an estimation using statistics collected previously about the underlying data. Statistics maintained about a database include: The number of tuples in a table, the number of unique values in a column, and the number of times a given value occurs. The number of I/O operations performed is used as the metric of cost as the time required to retrieve data from disk often dominates the time required for other processing.

2.2.3.3 Execution Engine

The execution engine is responsible for taking the physical plan generated by the optimizer, using it to answer the query by retrieving data from the storage engine, and executing the other algorithms specified. Traditionally, this is done by implementing each operator in the physical plan as an *iterator*. These iterators are then chained together to form an *iterator tree*.

An iterator is responsible for implementing the methods `open`[3], `getNext`, and `close`. The primary advantage to implementing each physical operator as an iterator is composability. Since each operator presents the same interface, they can be ordered arbitrarily, greatly reducing the number of cases that need to be implemented for each physical operator.

While each operator produces only a single tuple at a time, work can be batched within the operator for purposes of efficiency. For example, a physical operator that is performing a table scan could read each tuple from disk one at a time. However, this strategy could result in significant delays due to the need to seek the disk to the correct position for each read. Instead, tuples are generally read one page at a time and buffered, so they are ready to return the next time `getNext` is called. Another read from the disk is performed only when the buffer is empty.

Another related detail about the implementation of different operators is whether they are *streaming* or *blocking*. A streaming operator is capable of producing tuples without reading in all of the data from its child operator. For example, an operator that is performing a selection only needs to look at a single tuple at a time to decide if it matches the specified predicate. Thus, the selection operator can produce a result tuple as soon as a single matching tuple is read from the child iterator.

In contrast, a blocking operator such as sort needs to see all possible input tuples before producing any results. To understand why this is the case, consider returning a tuple before all input tuples have been read. It is always possible the next tuple read in will be less than the emitted tuple and thus, this implementation of the sort operator would have returned results out of sorted order.

2.2.3.4 Storage Engine

The storage engine lies at the bottom of the architecture stack and provides primitives for storing and retrieving data for the layers above. Generally, the responsibilities of this layer include reading and writing data from persistent storage, maintaining a buffer pool of commonly accessed data, performing logging so data is not lost during a failure, and ensuring that concurrent transactions do not conflict.

In a traditional system, the storage engine provides concurrency control primitives that allow the higher layers to provide the illusion of isolated execution to the programmer in spite of the presence of concurrently executing transactions. For example, the storage engine

[3]The `Typewriter font` is used for method names and other instances where the text refers to code.

could provide the ability to place read or write locks on different pages of data that prevent other transactions from touching that data.

The problem of concurrency control is much harder in a distributed system for two reasons. First, the latency between machines is much higher than within a single machine. As a result, the cost of holding locks can often be prohibitively expensive. Additionally, in a distributed system it is common for failures to occur in individual nodes without causing the entire system to fail. Without strongly synchronized clocks, it can be very difficult to reason about the state of locks held by nodes that have failed.

Due to these complications, most of the target applications forgo full ACID transactions and instead tolerate only eventual consistency. Thus, the storage manager of this database system will not be required to provide general locking. Chapter 6 talks about the concurrency primitives that are acceptable in a scale independent system.

2.2.4 Additional SQL Features

All modern relational engines offer additional features on top of what is defined by pure relational algebra. While a full discussion of all features available is out of the scope of this thesis, it is important to highlight those that directly impact the set of queries can be executed in a scalable manner. Specifically, the PIQL system leverages modifications to the way traditional RDBMS allow for the retrieval of partial results for a query. PIQL also gives developers new ways to enforce constraints on the data that is allowed to be stored in a given database.

2.2.4.1 Partial Query Results

It is common for developers to want only partial results for a given query. For example, a website like Amazon might want to display the top ten most popular items. Another case where partial results can be useful occurs when the total result set is too large. In this case, it is desirable to *paginate* the results, by displaying a subset of the full answer one page at a time. While it is possible to simply retrieve the entire result from the database and then only display a subset to the user, this can be very inefficient as it wastes a lot of work retrieving results that will never be displayed. For this reason, many modern database systems provide mechanisms for specifying what subset of a query result is desired. Examples of such mechanisms include the following:

Cursors Early database systems allowed the retrieval of partial results using a feature known as *cursors*. A cursor returns the results of a query one at a time. If the cursor is closed before retrieving all of the results of a query, then processing can stop early. However, this technique can still result in wasted work. For example, if the query contains a blocking operator such as a sort, then all tuples will need to be processed before a single tuple is returned by the cursor.

LIMIT Clauses The efficiency of query processing can be significantly improved if the database system is made aware of the fact that only some of the result tuples are desired. Carey and Kossmann suggested providing this information through the creation of the `LIMIT` clause, originally named `STOP AFTER`. Adding `LIMIT K` to a query tells the execution engine to return only the first `K` tuples.

The optimizer is made aware of this limit through the addition of a *stop operator* to the resulting logical query plan. A stop operator is a simple operator that returns the specified number of tuples before stopping. In order to reduce the amount of work required, the stop operator can be pushed down in the plan to reduce the number of tuples produced early.

The rules for how far a stop operator can be pushed down in a given query plan depend on whether or not restarting the query is allowed. Restarting a query can potentially be expensive, but it allows the stop operator to be pushed deeper in the plan. In contrast, if restarting the query is not allowed it is important that the stop operator not be pushed beneath any reductive predicates. A reductive predicate is one that can potentially reduce the number of tuples returned and thus subsequently cause the query to return fewer than K tuples even though more may exist in the database.

OFFSET The `LIMIT` clause is sufficient for running top-k queries and returning the results for the first page of a paginated query, but it does not allow for the efficient retrieval of tuples from the middle of the result set for subsequent pages. For this reason, many database systems also allow developers to specify an `OFFSET` in addition to a `LIMIT` clause, which causes the execution engine to skip the specified number of tuples before beginning to return the result. However, this mechanism does not avoid any processing that was required to produce the skipped tuples. Thus, using this method to produce pages one at a time requires a quadratic amount of work relative to the number of pages produced.

To understand why a quadratic amount of computation is required, consider returning the nth page with k results on each page. Producing the first page requires computing the first k tuples. However, when producing the second page those same tuples will be computed only to be skipped, thus the cost of producing the second page is $2k$. Therefore, the cost of producing the pages 1 through n is:

$$\sum_{i=1}^{n} ik = O(n^2) \tag{2.3}$$

2.2.4.2 Integrity Constraints

Many SQL systems also allow for the specification of *integrity constraints* as part of the schema of an application. The constraints are enforced when data is inserted, modified, or deleted and can enable the use of certain kinds of query optimization in addition to ensuring data integrity.

One example of a constraint that can be specified is a *uniqueness constraint*, such as that enforced for the *primary key* of a relation. This ensures that any given value for the specified attribute can exist in the database at most once.

Another type of commonly used constraint is a *foreign key constraint*. This type of constraint ensures referential integrity for data, meaning that, for example, it will prevent the insertion of data that refers to a nonexistent key.

Section 3.5.2 discusses a new type of integrity constraint, known as a *cardinality constraint*, that PIQL uses to increase the space of queries that can be guaranteed scale independent.

2.3 Materialized Views

Views are an important abstraction provided by database systems that help enable logical data independence. Using views, developers can present multiple representations of a given set of relations without having to store and maintain redundant copies of the data. A view is essentially a virtual table defined by SQL statements. Since views themselves are just a collection of tuples, developers can run SQL statements over them as though they were normal relations.

Often, views are a purely virtual abstraction. A virtual view is evaluated on-demand when a query is executed. Since a view is just a relational algebra expression, this evaluation can be optimized along with the final query. However, this computation can still be very expensive to evaluate over and over again each time a query is run.

To avoid such redundant computation, many database systems have support for *materialized views*. Instead of computing the result of the view expression each time a query is run, a materialized view is calculated once and then stored to disk. Since materialized views contain precomputed results, they can often be used to dramatically speed up query execution. Techniques even exist to automatically decide when using the results stored in a materialized view might help execute a query over the base relations faster [36].

However, since the contents of the view are stored and reused, the database now needs to be concerned with updating these cached results when the data stored in the base relations change. The simplest strategy to ensure that a view is up-to-date is to recompute the view from scratch each time any of the underlying data changes. This technique can be very expensive, however, especially for applications that have a very high rate of data modification. Fortunately, for many views it's possible to be more efficient by updating the view *incrementally*, only recomputing the rows that change.

In Chapter 5, I describe techniques that leverage materialized views to enable scale independent execution of a wider range of queries than would be possible without precomputation. To enable this discussion, the remainder of this section is devoted to a review of relevant prior work on materialized views, which will serve as the basis for the a scale-independent view selection system. Generally, this work falls into two categories. The first, *view selection*, refers to the problem of deciding which views should be materialized by the system.

The second, *view maintenance* is concerned with the efficient updating of materialized views given an update to the base relations.

2.3.1 Automatic Selection

While a smart database administrator (DBA) can often manually decide which materialized views will help speed up query processing, it is also possible for the system to do this automatically. Theoretically, this problem can be framed as an optimization problem of the following tuple (S, V, M, Q) [67]. S represents the schema of an application along with a size estimation for each relation. V represents the set of all possible views over the schema S. M is the amount of space available in the system for the materialization of views. Q is the set of queries present in the workload. A standard view selection system attempts to find the best subset of V that fits in the storage M such that the queries in Q execute as quickly as possible.

Static Selection Early work on materializing *data cubes* [38], a particular type of materialized view, performed view selection statically. A data cube is a multi-dimensional structure that enables the efficient execution of aggregation queries over a variety of data groupings. To better understand the types of queries that might be sped up by a data cube, consider the TPC-D benchmark [82]. TPC-D simulates a decision support workload for a manufacturing company. Aggregate queries for this benchmark commonly group over three different attributes: `part`, `supplier`, and `customer` of the relation LineItems.

When deciding which views to materialize, it is important to realize that some views could be used to answer queries against other views, albeit less efficiently. Specifically, a view whose grouping attributes are a superset of another view's grouping attributes can always be used to answer queries against the latter view. For example, a query that groups on `part` could be answered by view grouped on `part` and `supplier` by further aggregation of tuples with the same value for the `part` attribute. However, the converse is not possible, as it is impossible to "un-aggregate" values that have already been grouped together.

To help reason about the trade-off between materialized view size and query execution time, Harinarayan et al. [44] suggested representing the set of possible views as a lattice. Each vertex in the lattice represents a possible materialized view, and edges represent which views can be used to satisfy queries against others. Figure 2.4 shows the lattice structure for the TPC-D schema along with the estimated number of rows for each possible materialized view. This lattice could be extended further by including a date field present in the schema. Date fields also contribute to the lattice structure in a manner similar to that of the subset/superset relationship of group attributes. Specifically, data aggregated over smaller time units (e.g., days) can be used to calculate queries over larger time units (e.g., years).

By making the assumption that query execution time is proportional to the number of rows read (an assumption that has been validated experimentally [44]), it is now possible to reason about the optimal set of views given a set of queries and a space constraint. Finding the solution to this optimization problem however is NP-Complete, as solving it is equivalent

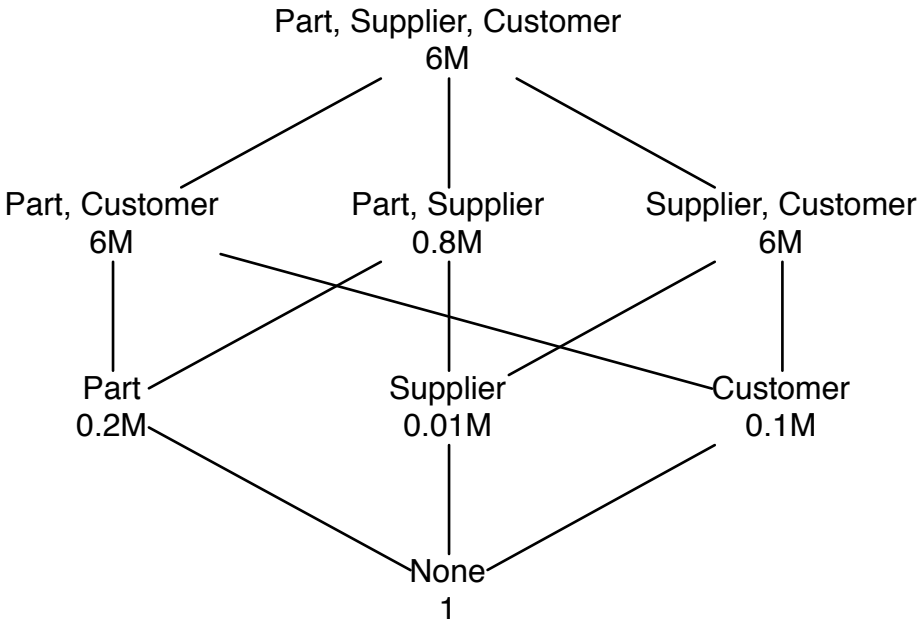

Figure 2.4: The lattice of possible materialized views for the TPC-D schema. Adapted from Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. "Implementing data cubes efficiently". In: *SIGMOD Rec.* 25.2 (June 1996), pp. 205–216

to solving the set-cover problem. Fortunately, there is a greedy algorithm [44] that produces answers very close to optimal and which has been shown experimentally to provide good performance.

Other work has since extended this type of static view selection to consider the cost of view maintenance in addition to storage space [15, 42, 56, 70, 90], though discussion of these techniques is out of the scope of this thesis.

Selection Based on Workload For more complicated schemas and workloads, it is not feasible to enumerate all possible materialized views. Instead, Agrawal and others [4] suggest a view selection system that starts by taking as input a trace of the workload from a production database. From this trace it is possible to enumerate *syntactically relevant* views and indexes for each query present.

To understand the concept of syntactical relevance, consider the following query over a schema similar to the TPC-D schema described above.

```
SELECT SUM(qty) FROM LineItems WHERE part='transmission'
```

The following queries are among those that are syntactically relevant to the above query, as they could be used to compute the answer more efficiently.

```
SELECT SUM(qty) FROM LineItem WHERE part='transmission'
SELECT SUM(qty) FROM LineItem GROUP BY part
SELECT SUM(qty) FROM LineItem GROUP BY part, supplier
SELECT SUM(qty) FROM LineItem GROUP BY part, customer
SELECT SUM(qty) FROM LineItem GROUP BY part, customer, supplier
```

A key problem with this approach is selecting *candidate materialized views*. Specifically, it is important to avoid considering all syntactically relevant queries, as this would quickly result in far too many candidate views to consider efficiently. Instead, Agarwal et al. take a three part approach based on the concept of *table-subsets*, a subset of the tables present in the query from the workload. Given a query from the workload, the view selector starts by picking a useful subset of all possible table-subsets for the query. The metric for determining if a given table-subset is useful is based on how often it appears in other queries from the workload. Next, the algorithm proposes a set of possible materialized views based on the useful table-subsets. A cost-based analysis is used to select the best possible materialized views to include in this set. Finally, starting with all of the possible views from the previous set, a "merged" set of views is created that can be used to satisfy multiple queries from the workload.

2.3.2 Incremental Maintenance

It is often possible to update a materialized view incrementally when the database is modified instead of recomputing the whole view from scratch. The problem of incremental view maintenance has been widely studied and early techniques differed in several dimensions [41] including:

Information Dimension What data is available to the algorithm while updating the view? Specifically, can the base relations or the current contents of the view be accessed while performing the update? Views that can be updated without reading the base relations are called *self-maintainable views*. Such views can be especially advantageous in distributed data warehousing environments. In this distributed scenario, data is often spread across multiple servers. Self-maintainable views can be updated on a single machine without needing to perform long read transactions against base relations that may reside on another machine, thus decreasing contention and improving performance.

Modification Dimension What types of updates can the maintenance algorithm handle? Insertions? Deletions? Updates? Are updates handled as a deletion followed by an insertion?

Language Dimension What types of queries are supported? Some algorithms are only capable of handling select-project-join queries, while others are can handle more complicated expressions including aggregation, recursion, duplicates, and so on.

Instance Dimension Does the view maintenance algorithm work for all instances of the database and all possible modifications or not?

For the purpose of this thesis, I focus on views that can be updated given access to the base relation (i.e., not self-maintainable), have no duplicates, but might include aggregation. Ceri and Widom [21] proposed a method for incrementally maintaining such views – minus aggregation – using production rules. Production rules are implemented using a common database system feature known as a *trigger*. Triggers allow a developer to specify rules that will execute anytime the data in a relation is modified. At a high level, Ceri and Widom's view maintenance technique is based on the idea of *delta queries*, which compute the tuples that must be added or removed from the view given the insertion or deletion of a single tuple from a base relation. Updates to a tuple can be handled by running the rules for a deletion of the original tuple followed by an insertion of the updated tuple.

However, before the delta queries can be calculated, the system must first ensure that efficient incremental maintenance is possible by performing a static analysis of the view definition. Efficient maintenance using this technique requires the view to satisfy two conditions:

Duplicates The view must not contain duplicate tuples, though extensions that utilize a counting solution allow this requirement to be relaxed [39].

Safe References All references to base tables in the view must be "safe". Safety occurs when all known keys for each base table are present in the view definition. Note that a key attribute does not necessarily need to be present in the projection of the view for it to be implicitly specified by the data of the view. If there is an equality constraint with an absent key attribute and an attribute that is present in the view then the value of the absent key attribute is still considered specified. Thus, the check for safety must transitively figure out which attributes values are known, given a tuple from the view and the equality constraints present in the view definition.

For an insertion, the production rule executes the SQL statement that defines the view, replacing the modified table with the tuple being inserted, and inserts the results into the view. Deletions occur symmetrically. Updates are modeled as a deletion followed by an insertion. As an example, examine the following schema and view definition.

```
Products(productID, name)
LineItems(orderID, date, qty, productID)

CREATE MATERIALIZED VIEW Sales
SELECT orderID, qty, l.productID, name
FROM Products p, LineItems l
WHERE p.productID = l.productID
```

First, the algorithm verifies the possibility of efficient maintenance for this view definition. The following attributes are present or *bound* in the view: {`l.orderID`, `l.qty`, `l.productId`,

p.name}. The attribute p.productID can also be added to this set due to the equality constraint. Thus, the key attributes for both relations are present in the view definition and the view can be efficiently maintained. Next the following two production rules can be constructed from the view definition.

```
ON INSERT Products
INSERT INTO Sales
SELECT orderID, qty, productID, @name
WHERE @productID = productID

ON INSERT LineItems
INSERT INTO Sales
SELECT @orderID, @qty, @productID, name
FROM Products
WHERE productID = @productID
```

In the preceding example, `@attribute` denotes the value of the attribute for the tuple being inserted.

If there were an integrity constraint (described later in Section 2.2.4.2) ensuring that productIDs from LineItems were present in Products, then the first rule would be unnecessary as it could never produce any tuples. The rules for deletions are symmetric and thus omitted for brevity.

It is possible that, even after performing the substitution provided above, the resulting delta query can still be a complicated relational expression. Ahmad and Koch [5] realized that instead of evaluating these delta queries directly, the update can often be more efficiently computed through the creation of another materialized view. Creating this view can result in the recursive creation of many materialized views. Fortunately, it can be shown that for an interesting subset of SQL, this recursion is guaranteed to terminate [50], thus creating a finite set of views. The benefit of this recursive view creation approach is that the maintenance operations for each view are now simple queries that avoid complex join calculations. This property is exploited in Chapter 5 by the PIQL scale-independent materialized view system.

2.4 Building Large Scale Web Services

Large scale web sites present a particularly interesting data management challenge, one where existing techniques often fall short. While the ability to handle rapid growth without experiencing performance degradation is a useful property for developers in many domains, these web applications present a new problem for two primary reasons. First, their rate of growth is astronomical. As stated in Chapter 1, it is not uncommon for applications such as Twitter to experience years of sustained exponential growth [88]. Second, these applications are of increasing relevance to the everyday life of a large subset of the population. Additionally, the fact that these applications are often interactive in nature means that they

are especially sensitive to the performance problems that can occur as a result of rapid growth.

In this section, I explain some of the most common tools that are currently used to build and scale these applications. In addition, I focus on how developers currently attempt to manage rapid growth, and I discuss some of the ways these technique remain inadequate.

2.4.1 History

Early web services were implemented as simple common gateway interface (CGI) scripts. CGI provides a common interface that allows developers to write programs that interface with webservers and dynamically generate web pages. CGI, however, only provides a very basic protocol and thus requires programmers to perform many common tasks manually. For example, developers would often need to deal with problems such as parsing responses to forms, maintaining state in-between different requests, as well as building web pages from templates.

In contrast to the simplicity of CGI, modern web programming frameworks are often designed with many more features and can help alleviate much of the redundant programming work. Many different web frameworks exist, with at least one being available for every major programming language. Some popular examples include Rails (Ruby) [71], Play (Scala) [63], and Django (Python) [32].

2.4.2 Architecture

Most web applications are implemented using a 3-tiered shared nothing architecture. Figure 2.5 illustrates an example of this architecture for a simple web application, where each box represents a separate machine. A key characteristic of this architecture is that there is typically no communication between the servers of any given tier.

Typically, requests come in from many clients to a single centralized load balancer. The load balancer is responsible for distributing these requests across a set of web servers in the presentation tier. This tier is typically responsible for simple static content such as images and is not discussed further. In the middle of this system lies the application tier, which is responsible for implementing the logic of the application. At the bottom of the stack is the data storage system, which is often a relational database system, as described in Section 2.2.

2.4.2.1 Application Tier

The actual code that comprises the website runs on the servers of the application tier. Typically, the application server is designed to be *stateless*. A stateless system is one that does not preserve any data across different requests.

There are many benefits to implementing the application code in a stateless manner. First, this design choice makes it much easier to handle load balancing and fault tolerance. Requests can be spread out in a round-robin fashion, without worrying about which server

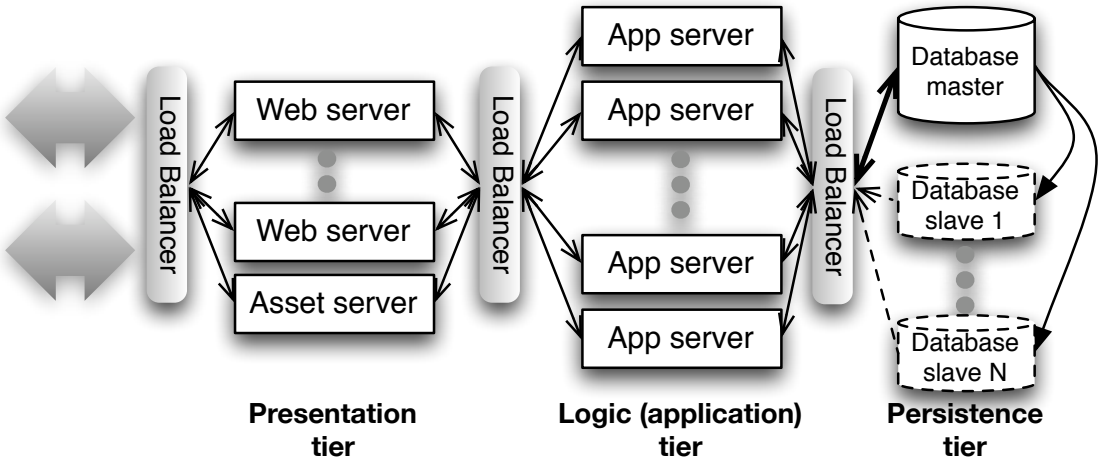

Figure 2.5: A 3-tiered shared nothing architecture for a simple web application. Reprinted with permission from Armando Fox and David A. Patterson. *Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Edition*. Strawberry Canyon LLC, 2012

handled previous requests from a particular client. Additionally, if a given application server fails, no recovery is necessary, and requests can simply be routed to a functioning server.

Finally, the stateless nature of the application server makes it very easy to scale the processing power at hand as demand for the application changes. When capacity is overprovisioned, servers can be removed from the pool as soon as they finish processing any outstanding requests. Similarly, when demand increases, servers can be added without needing to propagate any state information.

2.4.2.2 Data Tier

Since the application tier is implemented as a stateless service, all persistence across requests is often left up to the data tier. As a result, it is generally much more difficult to scale this part of the system as demand increases.

Unlike the application tier, which is commonly split up amongst many relatively inexpensive commodity nodes, the database system often runs on more expensive, specialized hardware. Thus, when demand begins to outpace capacity, the easiest option for growing the application is often to attempt to buy a larger machine for the database system. For many applications this solution is perfectly acceptable, albeit expensive. For very popular applications, however, this strategy eventually falls short once even the largest available machine is unable to handle the required request rate.

Once demand outstrips the capability of any single database machine, the developers must begin to consider *partitioning* the database across many nodes. Often, developers faced with this challenge begin by *vertically partitioning* their system. A vertically-partitioned database

is one where different tables exist on different machines, but no single table is split across nodes. While this step is often relatively easy to implement, it does make performing joins of the tables that are now on separate machines more difficult.

For very popular sites, the vertical partitioning approach will also eventually be insufficient, and developers will be forced to consider implementing *horizontal partitioning* or *sharding*. In a horizontally-partitioned system, each relation is split across many different database servers. While some commercial database systems, such Microsoft SQLServer, provide native support for this type of partitioning, this support is less common in their open source equivalents. Hence, many sites must handle the routing of requests to the server responsible for the needed partitions manually in the application-level code. Performing this routing correctly often requires significant modifications to the way the application interacts with the database system. As a result, it can take a significant amount of time to modify a complex application to run against a database that has been partitioned. This difficulty causes implementation of this scaling strategy to be a frequent stumbling point for developers attempting to improve an application that is failing to handle the current workload.

However, many sites find that implementing database partitioning alone is not enough to deal with huge increases in traffic. In order to avoid expensive redundant computation and to further reduce the load on the database system, it is also common to add a caching tier to the application. Many popular websites — including Twitter, Flickr, and Wikipedia — use memcached [59] for this purpose. Memcached provides developers with a very simple interface to an in-memory key/value store.

Developers commonly use memcached to store the results to specific, expensive queries. This approach often significantly improves performance for several reasons. First, since memcached is all in-memory, the cost of seeking the disk arm to the correct location, and its associated latency, can be avoided. Second, the use of the cache allows much data retrieval to be performed without touching the database system at all. This ability to answer common queries without the database system can significantly reduce the load on this critical component and thus improve performance of the whole application significantly.

2.4.3 Service Level Objectives

Interactive applications are characterized by the need for fast response time for all operations. As mentioned in Chapter 1, even delays as small as 200 ms can have a significant effect on user behavior [73]. Unfortunately, the distributed nature of their architecture makes diagnosing performance problems even more difficult than in a centralized system. Not only are the various tiers and services running on different sets of machines, but they are often implemented by different teams of programmers. To further complicate things, many different systems might all rely on common subsystems in which one aggressive consumer of resources can hurt the performance of unrelated components.

As a result, it is common to monitor the performance of each component carefully and to express the expected performance as a service level objective (SLO). However, system architects and developers need to select a useful set of metrics. Considering only the average

performance could mean that a significant number of users are still experiencing unacceptable performance. Due to the use of large quantities of commodity hardware, however, looking at the worst case performance can be equally ineffective, since fixing the worst case performance in such a complex, failure-prone system is virtually impossible. Instead, developers of these applications often use *high quantile* response time as the indicator of application performance. For example, Amazon typically sets the performance goals for their applications relative to the 99.9th percentile response time, based on a cost-benefit analysis [31]. A typical SLO for a web application would specify that 99.9 percent of requests will complete in 300 ms or less.

2.5 NoSQL Storage Systems

As more websites began to struggle with the challenge of scaling traditional relational databases, a new class of storage systems began to emerge. Examples of NoSQL storage systems include Google's BigTable [23], Amazon's Dynamo [31], Apache Casandra [77], amongst others. While the design of these systems was often inspired by research on distributed hash tables (DHTs), they often involve a number of architectural simplifications that are made possible by the relatively controlled environment of the datacenter. This trend towards large scale distributed storage systems became increasingly popular after Google and Amazon began discussing publicly the details of their architecture.

The distributed nature of NoSQL systems is of primary importance, although the name NoSQL seems to characterize these systems by their sacrifice of a query interface in the name of scalability. Since these system were designed to take advantage of many machines, they automatically handle many distributed computing concerns, such as routing requests and rebalancing data if a single machine gets overloaded. Additionally, a common primary design goal of NoSQL systems is the ability to provide consistent performance even as the load on the system increases significantly. For example, Amazon reported that, through a combination of replication and judicious over-provisioning, their system Dynamo can provide a 99.9th percentile response time of under 300ms even during the Christmas shopping season [31].

The space of systems that classify themselves as "NoSQL" is large and rather amorphous. Therefore, this section will focus on qualities that will ensure scalability, rather than attempting to provide a full overview of possible design dimensions. Further discussion of the specific design choices that were made when implementing PIQL on a NoSQL storage system can be found in Chapter 6.

2.5.1 Data Models

NoSQL storage systems support a wide range of data models, including everything from relations to opaque blobs to full documents. For the purpose of scalability, the choice of data model is most relevant with respect to the level of insight the storage system has into

a given data item and how this insight affects the types of lookups that the system can provide.

At the simplest end of the NoSQL data model spectrum lie key/value stores that manage opaque arrays of bytes. Since the storage system has no knowledge of the encoding used to store the data, the types of operations that can be efficiently supported are relatively limited. Typically, such systems provide two main storage operations: `get`, which takes a key and returns a value; and `put`, which takes a key value pair and adds it to the storage system. The bytes that comprise the key can be hashed, allowing for an efficient implementation of both of these operations. This interface can enable the implementation of SQL constructs such as equality predicates over the key, but it will not allow the evaluation of range predicates or ordering constraints.

A more powerful data model allows for the efficient execution of range queries. One example of such a system is the column family model provided by the BigTable storage system [23]. The columns within any given column family are sorted lexicographically and can be enumerated by the client. This construct allows the efficient implementation of a relational engine that supports range queries, as discussed in Chapter 6.

2.5.2 Data Distribution

While some NoSQL storage systems, such as CouchDB[6], use only a single node, these are not particularly useful for scaling a large website. Those that natively support multi-node operation must decide how to distribute data across the machines in the cluster. The two dimensions of this decision are determining what partitioning function should be used and analyzing how to balance the use of replication versus partitioning.

The two most popular partitioning schemes are *hash partitioning* and *range partitioning*. Similar to the differences between hash indexes and B-Trees, the choice of partitioning scheme can affect what sorts of predicates can be evaluated without performing a full table scan. Hash partitioning is easier and trivially ensures that data will be distributed evenly. However, it only allows lookups by exact equality of the key used to store the data. Range partitioning, on the other hand, can suffer from hotspots due to popular key values, but allows for efficient order traversal by key.

2.5.3 Consistency

Traditionally database systems support Atomic, Consistent, Isolated, and Durable (ACID) transactions, which provide the illusion that only one query is running at a time. However, providing such transactions can be much more difficult in a distributed system. This difficulty is due both to the higher communication latency between nodes in the system, as well as the possibility of different components of the system failing at any given time. As a result, most NoSQL storage systems provide significantly weaker consistency guarantees than their relational counterparts. Fortunately, this weakening has proven to be an acceptable trade-off for many interactive web applications [86].

However, "weaker" consistency does not imply the total absence of consistency guarantees. Virtually all systems will promise at least *eventual consistency*. An eventually consistent system is guaranteed to converge on a single value for each key after an arbitrary amount of time has passed. While the amount of time is generally unspecified, stochastic models can often provide insight to the expected time required for convergence to occur [12].

Some systems will also provide write monotonicity with respect to a single key. This property allows the developer to understand the total ordering of writes, but only as it affects a single key. Providing this type of consistency is easier than general transactions, as generally achieving it requires less coordination amongst machines.

2.6 Summary

The task of writing data driven applications was greatly simplified by the introduction of the relational model. In particular, this model enabled the creation of declarative languages, such as SQL, which separate the specification of *what* data should be retrieved from *how* the specific execution should occur. This separation not only simplifies the task of the programmer, but gives flexibility to the underlying storage system to perform complex optimizations. Unfortunately, these techniques have often proven insufficient for the rapid growth and stringent SLAs that are typical of large-scale web applications, thus leading to the creation of many popular NoSQL storage system. In the remainder of this thesis, I present scale independence, which allows developers to preserve the productivity benefits of the relational model, and still be assured that their applications will perform predictably as data sizes grow.

Chapter 3

A New Data Independence: Scale Independence

3.1 Introduction

As described in the previous chapter, the data independence provided by relational database systems can be a huge boon to developer productivity. Separating the concerns of data management and retrieval from the specifics of the application allows developers to be more productive when creating the first version of their application. Arguably even more important, this independence enables greater agility by making it easier to add features that might require changing the schema of the application after it has gone public.

Unfortunately, data-independence itself can also make it very difficult to reason about the performance of the application, especially as the size of the database grows. Since the actual execution strategy is divorced from the query specified by the developer, it is possible for a new physical query plan to be selected based on updated statistics about the data stored. While these changes should in theory always result in better average performance, they can also result in unforeseen performance problems simply due to the amount of data involved. As a result, these unexpected execution choices can result in SLO violations in production.

As mentioned in Chapter 1, this unpredictability has led many developers to abandon relational database systems in favor of less fully-featured, but more predictable, key/value stores. While this technological shift often makes it easier to reason about the scaling behavior of an application, it also forces developers to reinvent many of the helpful abstractions that were previously provided by the database. This reinvention represents a significant development challenge, and can increase the time-to-market for new features.

Fortunately, the performance opacity inflicted upon developers by current relational database systems is not inherent to the relational model. Therefore, instead of throwing out all useful abstractions provided by modern database systems, I propose the introduction of a new type of data independence, *scale independence*. A scale independent system

attempts to maintain consistent performance even as the data and workload grow by orders of magnitude.

One technique for ensuring this predictability is to maintain invariants on the operations[1] performed and resources required for all queries in an application. This chapter starts by describing different classes of queries based on the amount of data that must be touched during their execution. Next, it discusses the optimization techniques used for queries that can be executed on-demand, formalizing the invariant that ensures that a given query performs a bounded number of operations in the worst case. Building upon this foundation, I expand the discussion to include queries where scale-independent, on-demand execution is not possible. Fortunately, for many of theses queries, precomputation through the creation of an *incrementally-maintained materialized view (IMV)* (Section 2.3) can enable scale-independent execution. Since a scale independent system must ensure that these automatically created IMVs do not themselves threaten the performance of the application as it grows, restrictions are placed on the resources required for their storage and maintenance. Finally, I give an overview of the scale-independent workflow, which can be used to analyze all queries in an application and determine which class each falls into, producing a list of indexes and IMVs that will enable scale-independent execution.

3.2 Data Scaling Classes

Before going into the details of scale independence, it is useful to step back and consider the sources of scale *dependence* in interactive applications. Figure 3.1 shows that queries can be divided into four classes based on their performance scalability as the database size increases. These four classes are briefly described below.

Constant In the simplest case, the amount of data required to process a query is constant. For example, in a web shop, data needed to display a particular product or to show the profile of a particular user based on a unique ID is naturally limited regardless of how many products or users there are in the database. The optimizer knows about this bound since such a query must have an equality predicate against the primary key of the relevant relation. Other types of queries that fall into this class include queries with a fixed `LIMIT` that do not perform any joins or that only perform joins against a unique primary key.

Bounded A second class of query involves data that will grow as the site becomes more successful but that is naturally bounded. For example, in social networks, it is known that while people will gradually add more friends over time, the average person has around 150 "real" friends [45]. Setting a maximum friend limit of 5000 friends, as

[1]For the purpose of this thesis, the term *operations* is used to refer to data retrieval or modification that can be performed with nearly constant latency. For example, a bounded lookup given an index. Section 3.4 discusses the specific assumptions that are made in greater detail.

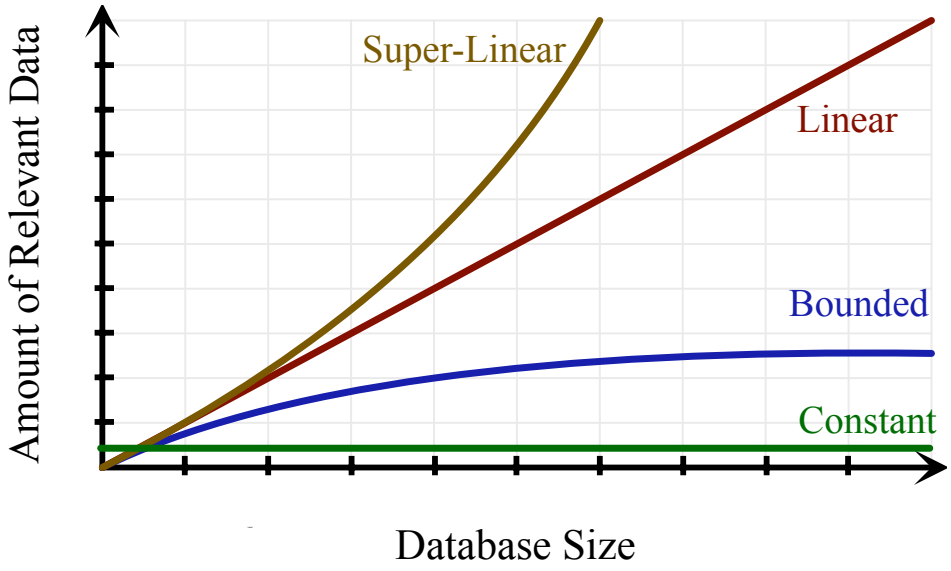

Figure 3.1: A comparison of the scalability of various queries as database size increases.

Facebook does, satisfies most customers [14]. A scale-independent system allows the developer to express these limits explicitly in the schema, through an extension to the DDL (see Section 3.5.2).

Sub-linear or Linear These queries require touching an amount of data that grows sub-linearly or linearly as the site becomes more successful. A query listing all currently logged-in users or a count over all customers falls into this category.

Super-linear These queries require computations over intermediate results that grow super-linearly with the number of users. For example, clustering algorithms that require computation of a self Cartesian product would fall into this class.

By definition, a success-tolerant web application can support only queries from the first two classes, constant and bounded. A scale independent storage system should identify such queries, and in the case of bounded queries, provide hints of acceptable cardinality constraints for meeting specific SLOs. Furthermore, if a bounded execution plan is only possible through precomputation in the form of secondary indexes or materialized views, a scale-independent system should automatically derive these requirements. For the unbounded queries from the second two classes, a scale independent system should notify the developer which portions of queries are unbounded and suggest workarounds, such as the introduction of cardinality constraints or the use of pagination.

3.3 Scale-Independent Execution Levels

A given query can fall into different query scaling classes, depending on the execution strategy that is chosen by the query optimizer. For example, executing a query using an index instead of a full table scan could result in that query being categorized as bounded instead of linear. However, executing this query using an index requires more resources when data is modified, as the index must be stored and maintained as data is added or removed.

In this thesis, I present a variety of techniques that ensure that a large number of complex queries can be executed scale-independently by touching only constant or bounded amounts of data. However, different techniques can require more resources as data is added or removed from the system. In order to help developers reason about how the use of these techniques will affect the resource requirements of their application, I define four levels of scale-independent query execution. While queries in all four levels can be executed at scale with predictable performance, different levels may require extra storage or computational resources in order to make this execution possible. Table 3.1 lists the four execution levels along with the invariants (Sections 3.3.1 and 3.3.2) on execution, update, and storage costs that are used to ensure scale-independance.

	SI-0	SI-1	SI-2	SI-3
Execution (Invariant 1)	I	D	D	D
Execution w/ sec. indexes (Invariant 1)	-	I	D	D
IMV Update (Invariant 2)	-	-	I	D
IMV Storage (Invariant 3)	-	-	I	I
IMV Parallel Updates (Relaxed Invariant 2)	-	-	-	I

Table 3.1: Levels of scale-independent query execution as defined by invariants on the query processing, update, and storage cost. 'I' and 'D' denote, respectively, that the cost of executing a query for a given resource is independent or dependent on scale of the application. A '-' implies the cost is not applicable and the query is trivially scale independent in this dimension.

The four scale-independent execution levels are as follows:

Scale Independence Level 0. The first execution level, SI-0, is capable of running trivially scale-independent queries (e.g., `SELECT 1`) as well as queries that can be answered in a scale-independent manner using only the clustered index on the primary key of a relation (e.g., a query that does a lookup by primary key).

Scale Independence Level 1.. For queries in SI-1, it is similarly possible to bound the amount of work performed while executing the query, but only through the use of a secondary index. While the required indexes can be created automatically, it is important for the developer to be aware of the increased storage and maintenance costs now associated with the execution of this query.

Section 3.3.1 provides a brief overview of the optimization techniques that statically analyze queries to determine if they can be executed under these first two execution levels. Chapter 4 provides an in-depth discussion of the algorithms used by my implementation, PIQL, while optimizing these queries.

Scale Independence Level 2 In contrast to the queries in SI-0 and SI-1, some queries could require an unbounded amount of work to execute on-demand, even after the creation of secondary indexes. Often, precomputation through the automatic creation of IMVs can fix this problem, enabling scale-independent execution by shifting work to insertion time. However, unlike with simple indexes, a scale-independent view selection system must also consider the scalability of storage and maintenance costs, and I reason about these costs through the use of additional invariants. Section 3.3.2 introduces these new invariants, which ensure that IMVs themselves do not become a scaling bottleneck. Queries can be executed under level SI-2 if they can be executed in a scale-independent manner using an IMV that satisfies both of the new invariants.

Scale Independence Level 3. For each of the aforementioned execution levels, the assumption is made that there exists a balanced partitioning of the workload over all of the machines of a parallel system. Section 3.4 explains in detail why this partitioning is required to avoid the increased query response time associated with workload hotspots. It is particularly important to be aware that the naive use of secondary indexes can violate this assumption in cases where there is temporal locality of insertions relative to the value being indexed (e.g., an index over a temporal attribute, such as `created_on`). Fortunately, these hotspots can often be mitigated by spreading new insertions across the cluster and periodically computing aggregate results in parallel. Since this execution pattern requires relaxing the invariant on the total work performed by an update, however, queries that utilize this strategy are executed under SI-3.

3.3.1 Scale-Independent Optimization

A scale independent system takes as input the set of all parameterized queries Q that will be used by an application. The optimizer can then analyze all queries in Q to ensure that the database can grow along the following three dimensions while maintaining consistent performance:

$\|R\|$	The size of all base relations,
Δ_{rate}	the update rate for all base relations, and
q_{rate}	the rate of read queries in the system.

A storage system can achieve scale independence across these dimensions by ensuring the optimizer will only select physical plans that perform a bounded number of storage operations, independent of $|R|$ (i.e., the size of all base relations). In contrast to standard average cost minimization performed by most relational database systems, this optimization technique prevents the selection of query plans that may perform well for most users but that could violate an application's Service Level Objective (SLO) for statistical outliers. It also allows the database system to warn the developer of queries that pose a potential scalability problem and to provide suggestions for resolving the issue before the query can cause SLO violations in production. The invariant maintained by this technique can be formalized as:

> **Invariant 1.** Let $Exec(q_i)$ denote the number of operations performed in the worst case by a query and $c_{ops}^{q_i}$ be a constant for a given query q_i. A scale-independent optimizer will only create physical query plans such that:
>
> $$\forall q_i \in Q \exists c_{ops}^{q_i} : Exec(q_i) < c_{ops}^{q_i}$$

A system can verify that an application will satisfy this invariant by performing a static analysis of the application's schema and queries. For example, uniqueness constraints ensure that a query that performs a lookup by primary key will return at most one result. Thus, as long as an index over the primary key is used to retrieve the matching tuple, this query will always require a bounded number of operations to execute.

To expand the space of queries that can be verified as scale independent, it is possible to employ language extensions to standard SQL. For example, PIQL introduces Data Definition Language (DDL) cardinality constraints, which allow developers to specify restrictions on the relationships present in their application. Similar to other types of data integrity constraints, these cardinality constraints are enforced at insertion time. To understand more concretely how this optimization technique allows a scale-independent system to bound the work required, consider a simple application that stores documents along with associated tags with the following schema:

```
Tags(docId, tag, timestamp) WITH CARDINALITY(tag, K)
Documents(docId, owner, timestamp, text, ...)
```

Italicised columns form the primary key of a relation and the `WITH CARDINALITY` clause denotes a cardinality constraint K on the number of unique values of `tag` that can exist for any given `docId`. An example of a scale-independent query on this schema is the following parametrized SQL, which returns the set of tags for a given document.

```
SELECT * FROM Tags WHERE docId = <doc>
```

For any value of the <doc> parameter, the query can be executed by scanning a bounded range of the clustered primary key index on Tags and will return at most K tuples. Thus, the optimizer can guarantee that this query will never violate Invariant 1.

Scale-independent queries do not necessarily need to execute against the clustered primary index. For example, consider the following query, which returns the text of the five most recent documents written by a given user.

```
SELECT text
FROM Documents
WHERE owner = <userId>
ORDER BY timestamp DESC
LIMIT 5
```

Without the presence of a secondary index, the only possible physical plans require a full table scan, and thus are not scale independent. As a solution, the optimizer can suggest executing the query under SI-1 through the creation of a secondary index over owner and timestamp.

3.3.2 Scale-Independent View Selection

Not all queries can be answered scale-independently using only indexes, and in this section I describe how IMV selection can enable the scale-independent execution of many of these previously unsafe queries. For example, consider the following query, referred to as twoTags, which returns the five most recent documents that are assigned two user-specified tags.

```
SELECT t1.docId
FROM tags t1, tags t2, documents d
WHERE t1.docId = t2.docId AND
      t1.docId = d.docId AND
      t1.tag = <tag1> AND
      t2.tag = <tag2>
ORDER BY d.timestamp
LIMIT 5
```

While a secondary index on Tags.tag would allow efficient lookup of the documents for a given tag, such an index is not sufficient to enable the scale-independent execution of the twoTags query. The performance of the query is potentially dependent on the size of the data due to the fact that during any given execution an unbounded number of rows matching tag1 might need to be scanned before finding five documents that also match tag2 or vice versa. In practice, developers faced with a query such as this one often utilize a technique known as *intersection precomputation* or *caching* [57], where all two-tag combinations for a document are computed ahead of time (analogous to the creation of a join index [85]).

As described in Section 2.3, there has been significant prior work on leveraging precomputation through automatic materialized view selection [3, 51, 60] and incremental maintenance [2, 16, 21, 40, 72]. These approaches, however, have focused on minimizing average cost for a given workload, rather than ensuring consistent resource requirements as the database grows. As such, these techniques could create a view that may speed up query execution on average, while the absolute performance of the query remains dependent on the size of the underlying data. Simply executing faster is not sufficient to guarantee SLO compliance as an application explodes in popularity. In contrast, a scale-independent system should creates IMVs when their existence will allow queries to be answered with a static upper bound on the amount of work that will be required for execution. In addition to determining the scalability of the query when it is run over the materialized view, the system must also ensure that the resources required to incrementally maintain and store the created materialized views do not themselves threaten the scalability of the application.

To better explain the potential scalability threat posed by the inclusion of materialized views, I now formalize the additional invariants that must be maintained by a scale-independent view selection system.

3.3.2.1 Bounding Update Cost

A scale-independent view selection system must avoid IMVs whose update cost increases with the scale of the application. To this end, a scale-independent optimizer must check that there is an upper bound on the number of operations required to incrementally update all indexes and views given an update to a single tuple in a base relation. Said formally:

> **Invariant 2.** Let $Update(r_i)$ denote the number of operations performed in the worst case by index and view maintenance when updating a single tuple in r_i, and let $c_{ops}^{r_i}$ be a constant for r_i.
>
> A scale-independent optimizer will only create views such that the total maintenance costs will never violate the following condition:

$$\forall r_i \in R \exists c_{ops}^{r_i} : Update(r_i) < c_{ops}^{r_i}$$

For queries in SI-3 it is necessary to slightly relax Invariant 2. This relaxation will permit queries where serial work performed by a single machine is bounded, instead of the total work done across all nodes for a single update.

3.3.2.2 Bounding Storage

It is possible for the size of an IMV to grow super-linearly with the size of the base relations, for example due to an unconstrained join. Therefore, a scale-independent system must verify that the size of each view can be at most a constant factor larger than one of the base relations present in the view. Said formally:

> **Invariant 3.** Let V be the set of all created views required to answer the queries in Q and let $c_{storage}^{v_i}$ be a constant for a view v_i. Let r_{v_i} be a relation in R and $|r_{v_i}|$ denote the

number of tuples in r_{v_i}. A scale independent system creates IMVs with linear storage requirements by ensuring that:

$$\forall v_i \in V \exists c_{storage}^{v_i}, r_{v_i} \in R : |v_i| < c_{storage}^{v_i} |r_{v_i}|$$

3.3.3 Query Compilation

My implementation of a scale-independent system, PIQL, includes an optimizer that only produces query plans that obey the three invariants listed in the previous section. Figure 3.2 shows the five phases of query optimization in PIQL. Together, these phases determine if scale-independent execution is possible for a given query. In addition to validating the scalability of all queries in an application and creating necessary indexes and IMVs, PIQL will also tell developers which execution level will be used for each query. This feedback gives developers the insight to reason about the query's resource requirements and update latency characteristics.

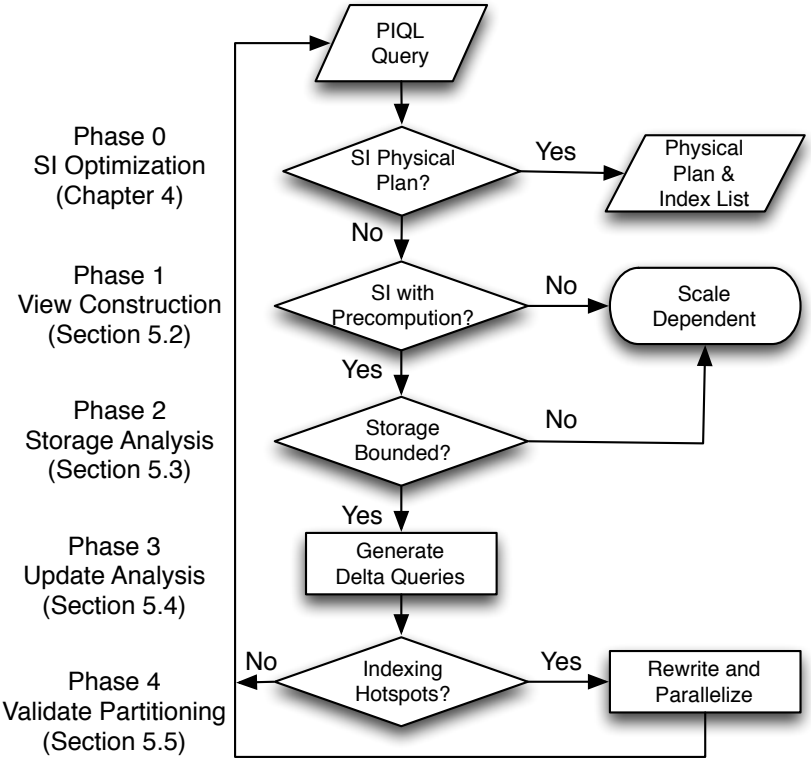

Figure 3.2: The phases of the PIQL scale-independent optimizer and view selection system.

Queries executed under SI-0 or SI-1 can be answered on demand by generating a scale-independent physical execution plan (Phase 0). Any indexes that are required should be automatically created by the system.

For cases where a scale-independent physical plan for a query cannot be found by the optimizer, PIQL will instead attempt to answer the query under SI-2, precomputing the answer through the creations of an IMV (Phases 1). Before creating this IMV, PIQL will check that the costs of maintenance and storage for this view will not threaten the scalablilty of the application (Phases 2-3).

Additionally, PIQL must ensure that common sources of hotspots are avoided (Phase 4). This phase attempts to rewrite the query to execute using a distributed staging step followed by a periodic parallel view refresh step. Since the resources required by the parallel view refresh step do not satisfy the strict form of Invariant 2, queries that require this transformation are considered to be executed under SI-3.

If at any point a query is determined to be scale-dependent, PIQL will invoke the Performance Insight Assistant (Section 4.4.4), which can help developers to identify and fix the problematic portions of the query.

3.4 Achieving Predictable Response Time

The bounds on execution cost enforced by Invariant 1 are useful not only for constraining resource requirements, but also for reasoning about performance. Specifically, PIQL attempts to provide scale-independent performance by taking advantage of the fact that many key/value stores can execute low-level storage operations operations such as `get(key)`, `getRange(prefix,limit)`, `put(key,value)` with consistent performance, even at high quantiles. For example, Amazon's Dynamo [31] demonstrated the ability to meet SLOs for get/put operations during their peak shopping season (December 2006) even in the 99.9th percentile on a large commodity cluster. Of course, a web application may experience load spikes in addition to normal fluctuation due to diurnal/seasonal usage patterns. However, recent research by Trushkowsky et al. [84] addresses this situation using an approach based on control theory in addition to well-known best practices such as replication and over-provisioning. Trushkowsky's experiments make use of the elasticity available in a cloud environment to scale a key/value store up or down in response to load changes while maintaining SLO compliance.

At a high level, PIQL is able to ensure consistent performance for all queries in an application by taking advantage of the fact that each query will only perform a bounded number of nearly constant-time operations in the worst case. Since each query or update performs only a bounded number of these operations, it is possible to reason about the probability distribution of the worst case execution time of each query. This worst case reasoning is especially valuable for developers of interactive applications who care about the response time for every user of their system. Taking this a step further, PIQL's SLO compliance model (Section 4.4) calculates the risk of violating a response time goal at scale using knowledge about the query plans and run-time statistics about PIQL's execution engine.

3.4.1 A Distributed Architecture for Scale Independence

PIQL is able to leverage the predictability of an existing distributed key/value store through a library-centric database system architecture. PIQL uses the key/value store as a record manager and provides all higher-level functionality (such as a declarative query language, relational execution engine, and secondary indexes) via a database system library. This approach is similar to the architecture employed by Google's Megastore, as well as by Brantner et al. [13, 17]. Figure 3.3) shows this architecture where each application server includes a PIQL database engine library that directly communicates with the key/value store. In accordance with the best practices discussed in Section 2.4.2, the application servers, and thus the database library, are designed to avoid preserving state between requests. This separation of the database system into a stateless component (the database library) and a stateful component (the key/value store) also serves to decrease the complexity of the PIQL system significantly. Query processing is performed at the client, thereby minimizing the functional complexity of the stateful component.

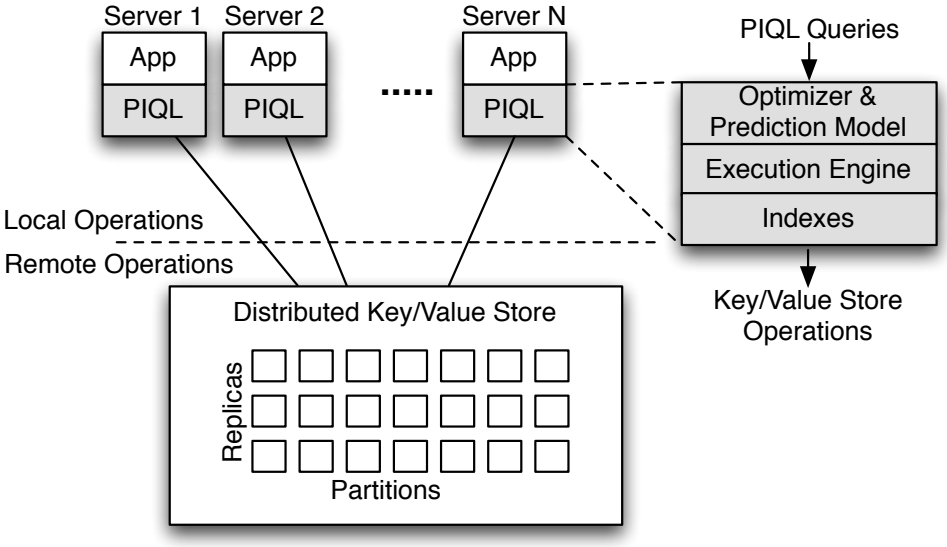

Figure 3.3: The PIQL database engine is implemented as a library that runs in the application tier and communicates with the underlying key/value store.

Much of PIQL's database engine is implemented using standard techniques. There are, however, several notable differences that arise as a result of the performance characteristics and limitations of the underlying key/value store.

Specifically, since remote operators retrieve data by issuing relatively high-latency requests against the key/value store, the trade-off between lazy evaluation and prefetching is even more severe than in a traditional system. Additionally, a key/value store, unlike a record manager running on a single machine, can support many requests made in par-

allel without interference. This distributed architecture enables two optimizations. First, the execution engine can leverage the cardinality information from the PIQL optimizer to prefetch all required data in a single request. Second, each of the operators is able to issue all of the requests to the key/value store in parallel. The experimental results, presented in Section 6.4.2.1, show that batching and parallelism greatly reduce the total response time for a given query.

3.4.2 Partitioning

While prior work [31, 84] demonstrated that it is possible to achieve nearly constant latency while scaling up, it is important to understand the implicit assumption that enables this predictability. Specifically, achieving consistent performance as the amount of data and number of machines grow is only possible when the growing workload can be spread evenly across machines in an ever-growing cluster. If no balanced partitioning of the workload exists, then eventually a single partition will become overloaded by the increased workload. As queues grow on the overloaded machine, the latency for requests to this server will grow and eventually affect the overall response time of the application. For example, a hotspot will eventually occur when there is a query that requires a secondary index over the creation timestamp of a given table. Maintaining this index naively would eventually result in a hotspot at the server holding the partition corresponding to the current time. Section 5.6 describes how PIQL automatically avoids the creation of such indexes, instead utilizing workload balancing coupled with a periodic parallel collection step.

3.4.3 Consistency

Finally, different types of consistency can have a significant effect on the ability of a storage system to provide predictable performance. Specifically, it is important that the mechanism used to provide consistency is nonblocking, as the variable performance that results from blocking operations, such as waiting for a lock, would violate the requirement of predictable performance for key/value store operations. PIQL's current implementation meets the non-blocking requirement by implementing eventual consistency. Section 6.6 provides more detail about the consistency semantics of the PIQL system. Alternatively, if a given application requires stronger consistency, it can be achieved using other non-blocking techniques such as snapshot isolation. A full discussion of how higher-level consistency guarantees can be implemented on top of key/value stores can be found in [17, 52].

3.5 Language Extensions for Scale Independence

A scale-independent system can increase the space of acceptable queries by extending standard SQL with constructs that allow developers to bound not only the number of results returned for each user interaction with the database system, but also to limit the cardinality

of intermediate results. In this section, I describe the DDL and DML extensions that enable the PIQL query compiler to bound the number of operations required even for complex queries involving joins or unbounded amounts of data.

3.5.1 Bounding Data Returned

In many applications, there are cases where the developer needs to run queries that would return potentially unbounded amounts of data, such as a query that lists all posts made by a user in chronological order. While this query itself cannot be made scale-independent, PIQL extends the SQL language to allow the developer to bound the number of key/value store operations required for each user interaction with the database by displaying subsets of the full result one page at a time. PIQL queries can contain a `PAGINATE` clause, which specifies how many items should be returned for each user interaction with the database system. Paginated queries are implemented as client-side cursors and can be invoked repeatedly, returning the next page of results each time. PIQL also supports the more traditional `LIMIT` clause for cases where only the top-K results of the query are required.

Additionally, to simplify the request routing and preserve the stateless nature of the application servers, the client-side cursor can be serialized and shipped to a user along with the results of the query. When the next page is desired, the serialized state is sent back to any application server where it is deserialized and execution can be resumed. The size of a serialized client-side cursor is generally small as only the last key returned by any uncompleted index scans needs to be remembered.

Note that the traditional methods of implementing pagination either require onerous server-side state management or are not scale-independent. Specifically, using server-side cursors requires the maintenance and garbage collection of the cursor state, even in the face of hundreds or thousands of users coming and going. The other common implementation of pagination uses both `OFFSET` and `LIMIT` clauses. Unfortunately, executing a query with an offset requires work proportional to the size of the offset (Section 2.2.4.1), which is in conflict with the goal of scale independence [37].

3.5.2 Bounding Intermediate Results

Standard SQL referential integrity constraints in the schema definition already allow the compiler to infer cardinality in one direction, from a foreign key to a single corresponding tuple. PIQL extends these constraints by allowing the expression of relationship cardinalities in the other direction as well. These developer-specified relationship cardinalities provide extra information for the optimizer and execution system about natural limits to the various relationships; these limits are often due to real-world constraints. The form of this specification is a maximum number of tuples that may contain a distinct value or set of values. For example, consider modifying the schema of Twitter to limit the number of other users that

any given person can "follow". This limit is expressed in the following schema:

```
CREATE TABLE Users (
  userId INT,
  firstName VARCHAR(255)
  ...
);

CREATE TABLE Following (
  ownerUserId INT,
  targetUserId INT,
  ...
  CARDINALITY LIMIT 100 (ownerUserId)
);
```

By specifying that there is a limit of 100 on the cardinality of any specific value of `ownerUserId` in the Following table, the developer informs the optimizer that no single user is allowed to have more than 100 subscriptions. Section 4.2 discusses in more detail how this limit is used during optimization.

Choosing an appropriate limit is crucial. As mentioned earlier, Facebook decided to use a very loose limit, 5000, for the number of friends. This looseness caused some power users with more than 3000 friends to complain about their experienced response times [74], as the system significantly slows down and some features completely break. In contrast, a Facebook competitor, Path, limits the number of friends to 50, which is even smaller than the natural limit. PIQL's prediction framework offers a tool to determine acceptable limits so that all queries meet the SLO requirements, independent of the scale of the system (see Section 4.4).

3.6 Summary

The difficulty of reasoning about the performance of declarative queries that are divorced from their actual execution plan by data independence has led many scale-oriented developers to abandon the helpful abstractions of the modern RDBMS in-favor of predictable but feature-anemic key/values stores. However, this sacrifice is unnecessary if it is possible to extend the relational model with the concept of scale independence. A scale-independent system gives developers the tools to reason about how the performance of their application will be affected by rapid growth, thus avoiding a possible "success-disaster".

A relational system can achieve scale independence by enforcing bounds on the number of operations that will be performed by any given query in an application, independent of the amount of data stored in the database. These bounds can be calculated statically before problematic queries can cause problems in production by using information from both the queries and the application schema. While such analysis is trivial for simple queries, PIQL is capable of running many complex queries, even those involving joins, in a scale-independent

manner. This increased expressivity is accomplished using a variety of advanced techniques including language extensions, automatic index creation, and precomputation.

Since the use of these techniques can affect the resource consumption and update-latency characteristics of the application, this thesis defines four levels of scale independent query execution. PIQL analyzes all of the queries in an application to ensure that they can be executed without violating the scale-independent invariants defined in this chapter. In addition, it informs the developer which execution level will be used for each query.

Chapter 4

On-demand Query Execution

4.1 Introduction

This chapter discusses PIQL's optimization strategy for queries that can be executed scale-independently without performing any precomputation. The execution of these "on-demand" queries falls into SI-0 (Section 3.3) if they don't require any extra indexes and SI-1 if secondary indexes must be created for their scale-independent execution. Central to PIQL's optimization strategy for these queries is selecting a physical plan that is guaranteed to perform a bounded number of operations independent of the size of the underlying database. This type of optimization can be contrasted with standard cost-based optimization, which attempts to find a plan that will perform the fewest operations. While running queries as fast as possible on average is generally beneficial, doing so does not take into account either the response time for statistical outliers or how response time will change as data is added to the system. Such variations in response time are significant concerns for website operators trying meet strict SLOs in order to keep their customer base happy.

Since performing a bounded number of operations is not particularly useful if the bound can be set arbitrarily high, Section 4.4 presents a model for reasoning about SLO compliance given various bounds on the number of operations. Using this model, developers can accurately predict if any of the queries in an application are likely to cause performance problems in production.

4.2 Scale-Independent Optimization

Given a PIQL query, the optimizer must select a scale-independent ordering of physical operators for its execution. A physical query plan is considered scale-independent if it is possible to statically ensure that it will perform a bounded number of simple operations in the worst case. Examples of simple operations include key/value store operations like a `get` by primary key or a `getRange` request that returns a bounded number of results. These operations are, however, examples based on PIQL's current implementation. Other storage

systems that are capable of executing further data retrieval operations with predictable response-time as data is added could easily be targets for this type of optimization as well.

To better understand PIQL's optimization strategy, consider one of the benchmark applications, SCADr. SCADr is a simple clone of the micro-blogging site Twitter[1]. The schema for the application is as follows:

```
Thoughts(userId, timestamp, text)
Subscriptions(owner, target)
```

SCADr allows users to post "Thoughts" (in Twitter terminology, "tweets") about their current activity. Users of SCADr can also subscribe to the thoughts of other users they are interested in, similar to a standard publish/subscribe system. The subscriptions for each user are stored as a mapping from the user who is subscribing (the `owner`) to the user whose thoughts are being subscribed to (the `target`). Throughout this chapter the "thoughtstream" query, which allows a user to retrieve the most recent "thoughts" of all the users to whom they are currently subscribed, is used as an example. Figure 4.1 shows the phases of optimization performed on the thoughtstream query.

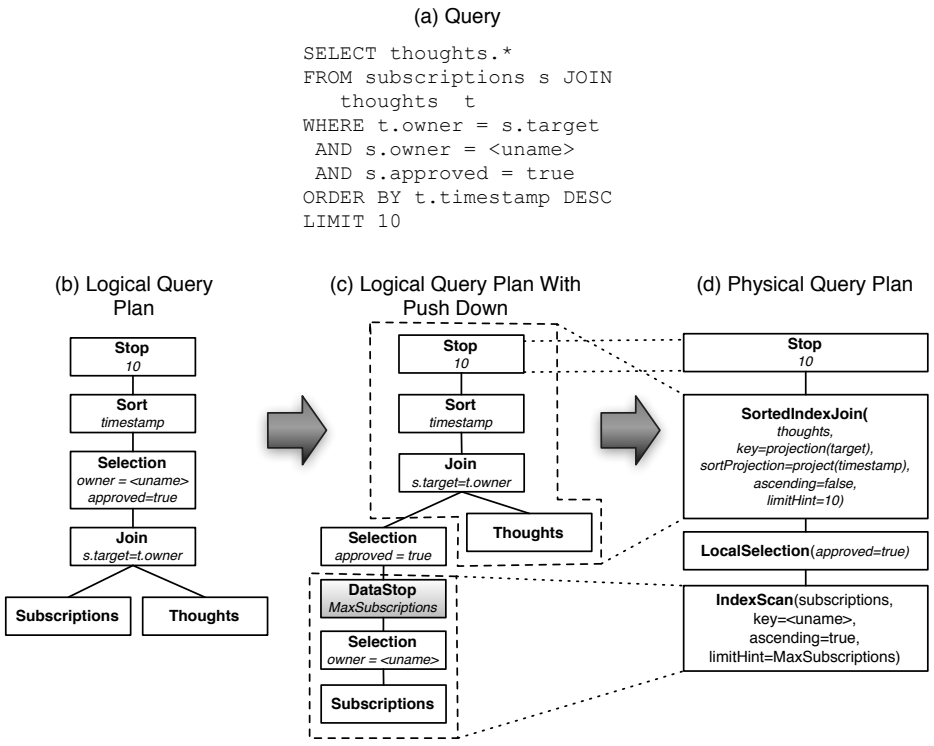

Figure 4.1: The stages of optimization for the thoughtstream query in SCADr.

[1]http://www.twitter.com

PIQL's optimizer operates in two phases as sketched in Algorithm 1 and Algorithm 2. In the following discussion, these two phases of the optimization algorithm are described in more detail. Special focus is paid to the scale-independent aspects of the query optimizer. Other optimizations, such as selecting join orderings, are performed using traditional techniques (e.g., [54]) and are therefore not discussed. Readers unfamiliar with standard query optimization techniques are referred to Section 2.2.3.2.

4.2.1 Phase I: Stop Operator Insertion

Given a logical plan from the query parser, Phase I starts by finding an appropriate linear join ordering (Line 1 in Algorithm 1). Next, the optimizer pushes predicates down in the plan using standard techniques (Line 2).

Algorithm 1 StopOperatorPrepare - Phase I

Require: $logicalPlan \leftarrow$ Logical PIQL Plan
Require: $cardinalityConstraints \leftarrow$ From the applications schema
1:
2: $orderedPlan \leftarrow findLinearJoinOrdering(logicalPlan)$
3: $preparedPlan \leftarrow predicatePushDown(orderedPlan)$
4: **for all** relation r in $preparedPlan$ **do**
5: **for all** combinations c of AttributeEquality predicates against r **do**
6: **if** $attributesOf(c)$ contains all fields in $primaryKey(r)$ **then**
7: Insert *data-stop* of cardinality 1 above c
8: **else if** $attributesOf(c)$ contains all fields of a cardinality constraint **then**
9: Insert *data-stop* of specified cardinality above c
10: **end if**
11: **end for**
12: **end for**
13: $finalLogicalPlan \leftarrow stopPushDown(preparedPlan)$ **return** finalLogicalPlan

Due to a `LIMIT` or `PAGINATE` clause in the actual query, the logical plan might already contain a standard *stop* operator to restrict the number of tuples returned [19]. Additionally, the optimizer will introduce new *data-stop* operators to the logical plan where schema-based cardinality information exists (Lines 3 to 11). The data-stop operator is a new operator that acts as an annotation, telling Phase II of the optimizer that a given section of the plan will produce no more than the specified number of tuples due to a schema cardinality constraint.

Any time equality predicates reference the entire primary key of the relation, a data-stop operator is inserted into the plan with a cardinality of one (Lines 5-6). The insertion of this data-stop operator is possible due to the uniqueness constraint on the primary key. This constraint ensures that there will never be more than one tuple in the database that satisfies these predicates. Data-stop operators are also inserted any time there are equality predicates present that reference all of the fields in a `CARDINALITY LIMIT`, specified as part

of the applications schema (Section 3.5.2). These data-stop operators are inserted into the logical plan with cardinality specified by the developer (Lines 7-8).

Afterwards, data-stop operators from the insertion phase as well as stop operators from a `LIMIT` or `PAGINATION` clause are pushed down as deep as possible into the plan (Line 12). Pushing stop operators deeper in the plan generally results in faster execution, as doing so limits the number of tuples that need to be processed earlier in the query's execution. While this optimization is useful for average performance, pushing down stop operators can also be useful when attempting to find a scale-independent execution plan for a query. Stop operators deep in the plan allow it to be broken into sections, each of which are guaranteed to perform a bounded number of operations, as described in the next section.

When performing the push down of the stop operators, it is important to ensure that the query can still be executed without restart. Executing a query that requires an arbitrary number of restarts could require performing an unbounded number of operations. Therefore PIQL performs this optimization conservatively, according to rules regarding non-reductive predicates [19]. Specifically, a stop operator cannot be pushed past a predicate that might reduce the number of tuples, as this could lead to an incorrect plan that produces fewer tuples than requested.

The rules for pushing down data-stop operators are different than those for standard stop operators. Recall that data-stop operators act as hints that are inserted based on the number of tuples that can possibly be stored in the database. Integrity constraints on cardinality, which are enforced at insertion time, ensure that no more than than the specified number of predicate-satisfying tuples will ever be present in the database. This constraint is in contrast to a standard stop operator, which merely indicates the number of results that are desired by the developer. Due to this difference, a data-stop operator can be pushed past all predicates other than those that caused its insertion. This degree of push-down is possible because even if a predicate reduces the number of tuples produced by the query, there cannot be any more tuples in the database due to the cardinality constraint.

Because the data-stop operator can be pushed further down in the plan than would be possible with a standard stop operator, it is possible to perform scale-independent static analysis of more queries. For example, in the optimization of the thoughtstream query, the data-stop operator is pushed past the predicate that ensures that a given subscription was approved. This degree of push-down would not have been possible with a standard stop operator. Since the optimizer is able to bound this section of the plan, its heuristic then chooses a local selection against the primary index instead of creating a new index that includes the approval field. This approach is cheaper both because it avoids maintaining an unnecessary index and because lookups over an index require an extra round trip to the key/value store to retrieve the full tuple.

4.2.2 Phase II: Physical Operator Selection

After the predicate and stop operator push-down, the optimizer transforms the logical plan recursively into a physical plan (Algorithm 2). This phase is effectively mapping the logical

operators of the query plan onto equivalent physical operators. The physical operators of the PIQL execution engine are broken into two groups: Those that operate locally on the client executing the query, and those that issue requests to the key/value store.

4.2.2.1 Remote Operator Matching

The operators supported by the current implementation of PIQL are based on those that can be performed with consistent performance as data volume grows by a key/value store, though this could be easily extended given another storage system capable of providing such predictability. In order to ensure scale independence, the optimizer requires that each section of the plan that will be executed using a remote physical operator has an explicit bound on the number of tuples required. This requirement is due to the fact that a `getRange` request will only have consistent performance if the number of tuples requested is bounded. Thus, whenever a plan section contains a group of logical operators that will be mapped to a remote operator, it must have either a stop operator or a foreign key uniqueness constraint. The enforcement of this restriction ensures that there will be a bound not only on the final result set, but also on any intermediate results that must be shipped across the network from the storage tier to the query processing library.

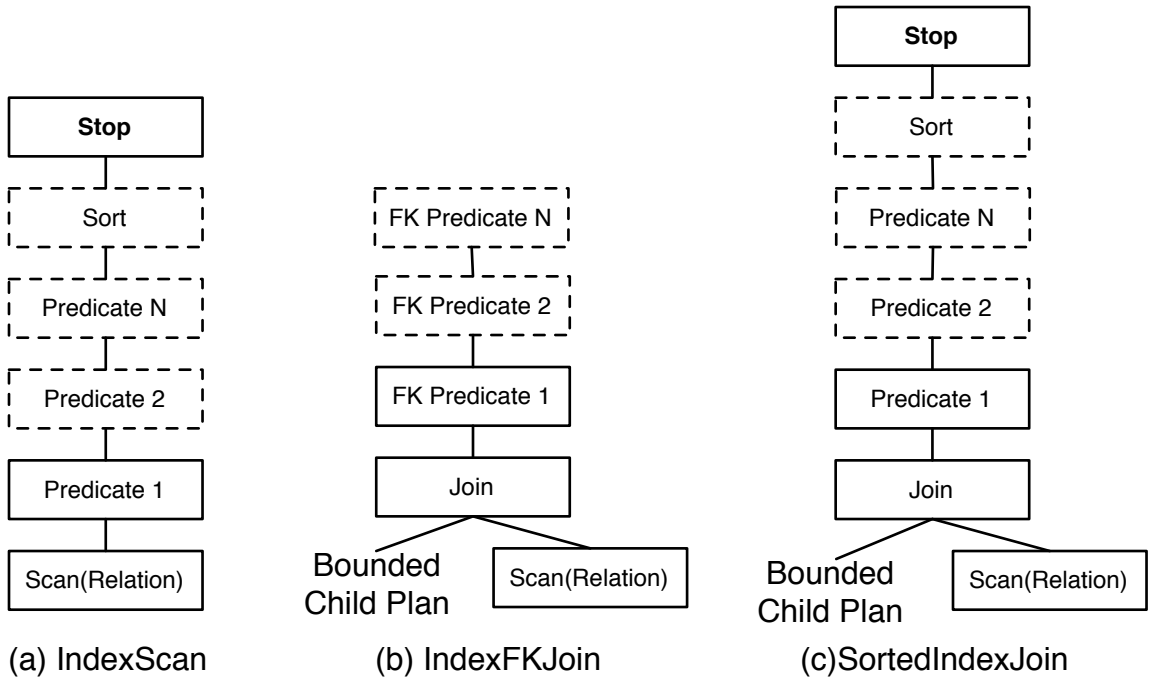

Figure 4.2: Every remote operator is equivalent to a pattern of logical operators. Optional logical operators are denoted by dotted line boxes.

PIQL supports three remote operators: *IndexScan*, *IndexFKJoin*, and *SortedIndexJoin*. Figure 4.2 shows the patterns of logical operators that can be executed by a given remote

operator. In the following, I describe the rules for mapping logical operators to the three remote operators.

Index Scan can be used for the physical execution of a set of predicates evaluated against a relation where the predicates describe some contiguous section of the index. Figure 4.2 (a) shows all the logical operators a single index scan can cover. In practice, this means that there can be any number of equality predicates, while predicates involving inequality may touch at most one attribute. Additionally, an index scan can be used to satisfy a logical sort operator using the special ordering of the index. However, if there is an attribute involved in an inequality, it must be the first field of any sort order to be satisfied by a scale-independent Index Scan; otherwise, this collection of predicates and sort constraints would by definition describe a potentially non-contiguous section of the index. Attempting to return potentially non-contiguous tuples from an index scan would make it impossible to bound the amount of work required to produce a given number of matching tuples. This impossibility is due to the fact that, even when tuples are not returned, the act of scanning over them requires them to be inspected and thus can still contribute to query processing time.

Index Foreign Key Join can be used for the physical execution of a join where the predicates filter for equality relative to the primary key of another table. Due to the uniqueness constraint of the primary key, the optimizer knows that the resulting number of tuples produced by the join will be less than or equal to the number of tuples produced by the child plan and thus the final result will also be bounded in size. Figure 4.2 (b) shows the logical operators that can be executing using the Index Foreign Key Join.

Sorted Index Join can be used for the physical execution of a join where there is an optional sort before the next available *limit hint*, as shown in Figure 4.2 (c). By using a composite index, the tuples can be pre-sorted for every join key and thereby leverage the knowledge of the limit hint to bound the number of data items per join key. For example, the thoughtstream query of Figure 4.1 (c) would normally require all thoughts per subscription. However, by pre-sorting the thoughts per subscription, the operator is able to retrieve only a bounded number (here 10) per subscription, which enables the overall bound.

4.2.2.2 Local Operator Matching

Operators that run in the application tier, called *local operators*, include sort, select, group by, and various aggregates. In contrast to remote operators, local operators work entirely on local data, which is shipped to the client. Consequently, as the remote operators ensure that all data is bounded in size, all local operators are bounded as well. The query language does not allow recursion; therefore, it is impossible for the local result size to increase infinitely.

4.2.2.3 Physical Plan Generation Algorithm

Algorithm 2 PlanGenerate - Phase II

Require: *logicalPlan* ← Logical PIQL Plan

```
 1: if (remoteType, child) ← match remote operator then
 2:     if logicalPlan has standardStopOperator then
 3:         return stop(remoteType(PlanGenerate(child)))
 4:     else
 5:         return remoteType(PlanGenerate(child))
 6:     end if
 7: else if (localType, child) ← match local operator then
 8:     return localType(PlanGenerate(child))
 9: else if logicalPlan = ∅ then
10:     return ∅
11: else
12:     ERROR(Not scale-independent)
13: end if
```

Algorithm 2 shows the general algorithm to transform a logical plan into an scale-independent physical plan. Starting from the top of the logical plan, the compiler tries to map as many logical operators as possible to a bounded physical remote operator according to the rules of Section 4.2.2.1 (Line 1). If the compiler finds a remote operator, it recursively calls the generator function with the remaining logical plan and attaches the resulting optimized child plan (Line 2-6). If no remote operator can be found, the compiler tries to find a local operator (Line 7) and if successful, it continues recursively (Line 8). If all logical operators are successfully matched either to a remote or a local operator, a bounded query plan has been found and it is returned (Line 10). However, if at any stage it is impossible to find either a remote or local operator, the physical plan could possibly require an unbounded amount of work during execution and is, therefore, not scale-independent (Line 12).

Figure 4.1 (c)-(d) shows an example transformation from a logical plan to a physical plan. The algorithm first selects an IndexScan to retrieve the subscriptions for a given owner. Note that this selection was made possible by the data-stop operator inserted as a result of the cardinality constraint on the number of subscriptions allowed for a given owner. Afterwards, it chooses a LocalSelection on the approved status, followed by a SortedIndexJoin and stop operator.

4.2.3 Index Selection

Since table scans are not scale-independent, as they require reading a linearly increasing amount of data (Section 3.2), the PIQL optimizer produces a list of all necessary indexes

during query optimization. These indexes can be automatically created by the system. For example, consider the following query from the TPC-W benchmark:

```
SELECT I_TITLE, I_ID, A_FNAME, A_LNAME
FROM ITEM, AUTHOR
WHERE I_A_ID = A_ID
AND I_TITLE LIKE [1: titleWord]
ORDER BY I_TITLE
LIMIT 50
```

The PIQL optimizer will select an IndexScan using an index consisting of the fields (token(I_TITLE), I_TITLE, I_ID) with a limit hint of 50. The first field allows the IndexScan to find all of the titles that contain the given token. The second field ensures that the items returned by taking the top 50 records from this index will be sorted by the full title of the item. Finally, the I_ID allows the execution engine to dereference the index and retrieve the actual item. Above this IndexScan, the optimizer would place a join with the author relation on the primary key A_ID.

4.3 Evaluation of Scale-Independent Plan Selection

My evaluation of PIQL's ability to maintain performance as an application grows is performed using queries from two different benchmarks, TPC-W and SCADr. This section begins with a description of each of these benchmarks.

4.3.1 TPC-W

TPC-W is a web application throughput benchmark for database systems[83]. It models an online bookstore with a mix of fourteen different kinds of requests such as searching for products, displaying products, and placing an order. Every request consists of one or more queries to render the corresponding web page. Furthermore, the TPC-W benchmark specifies three kinds of workload mixes: (a) browsing, (b) shopping, and (c) ordering. A workload mix specifies the probability for each kind of request. In all the experiments reported in this paper, the ordering mix is used because it is the most update-intensive mix (30% of all requests lead to an update).

The TPC-W benchmark measures the request throughput by means of emulated browsers (EBs). Each EB simulates one user who issues a request, waits for the answer, and then issues the next request after a specified waiting time. The TPC-W metric for throughput is Web Interactions Per Second (WIPS). According to the TPC-W specification, 90% of requests must meet the response time requirements. Depending on the kind of request, the allowed response time varies from 3 to 20 seconds.

The following experiments concentrate on the query execution part of TPC-W. Thus, the full web pages are not rendered, but instead, only the query execution portion of each web

interaction is performed. Since the response time requirements specified by the benchmark are in terms of end-to-end latency measured at the browser, these results are not directly comparable. However, since the query latency values are small compared to the given SLOs, query execution will clearly not be the cause of violations. Furthermore, while standard TPC-W requires full ACID guarantees, the current prototype implements only eventual consistency (described further in Section 6.6). Additionally, the wait time between requests was ignored, allowing the clients to place more load on the system with fewer machines.

Finally, the optimization techniques from this chapter are not sufficient to execute two of web interactions present in the TPC-W benchmark: AdminConfirm and BestSeller. These queries are analytic in nature and thus there is no scale-independent query plan that can be executed on-demand (i.e., using execution levels SI-0 or SI-1). However, Chapter 5 describes techniques for executing these queries scale independently through the use of precomputation.

4.3.2 SCADr

SCADr is a website that simulates the microblogging platform Twitter by allowing users to post "thoughts" of at most 140 characters [7]. Users can create a list of other users that they wish to follow, and the most recent thoughts from these users will be displayed in a thoughtstream when they log into the site.

The schema for this application is relatively simple and consists of three tables: Users, subscriptions, and thoughts. The users table contains a username as primary key as well as normal user attributes such as password and hometown. The subscriptions table specifies which users are subscribed to whom; that is, it models the n-to-m relationship between the users themselves. The primary key of the subscriptions table is composed of the owner of the subscriptions, followed by the target user. An additional attribute of the table specifies if the subscription has been approved. Finally, the thoughts table stores all the thoughts (i.e., microblog posts) of a user. The thoughts relation is composed of three attributes: Username, timestamp, and the actual message, which is limited to 140 characters. The primary key of the thoughts table is composed of the username and the timestamp of the thought.

The SCADr benchmark defines 5 different kinds of queries: "List users I'm following", "List my recent thoughts", "List the most recent thoughts of all of the people I am subscribed to", "Find user", and finally "Post a new thought", the only updating query. I measured both the request throughput and response time for executing all queries for a randomly selected user. This workload simulates a group of applications servers issuing database queries against the PIQL system, but not the page rendering portion of the site. Except for the "Post a new thought", which occurs with a probability of 1 percent, each of the remaining queries is executed once for every simulated request. The "Post a new thought" query is not further considered, as it is just a single put request.

4.3.3 Qualitative Analysis

The evaluation starts with qualitative analysis of the PIQL optimizer's ability to find scale-independent physical plans for all of the query in each benchmark. In cases where this is not initially possible, suggestions provided by the optimizer for improving scalability are implemented and discussed. Table 4.1 summarizes the necessary modifications (to either the query or the schema) for making the queries scale-independent, as well as the compiler selected indexes.

	Query	**Modifications**	**Additional Indexes**	SI Level
TPC-W Benchmark	Home WI	-	-	SI-0
	New Products WI	Tokenized search	Items(Token(I_SUBJECT), I_PUB_DATE)	SI-1
	Product Detail WI	-	-	SI-0
	Search By Author WI	Tokenized search	Authors(Token(A_FNAME, A_LNAME)), Items(I_A_ID, I_TITLE)	SI-1
	Search By Title WI	Tokenized search	Items(Token(I_TITLE), I_TITLE, I_A_ID)	SI-1
	Order Display WI Get Customer	-	-	SI-0
	Order Display WI Get Last Order	-	Orders(O_C_UNAME, O_DATE_TIME)	SI-1
	Order Display WI Get OrderLines	-	-	SI-0
	Buy Request WI	-	-	SI-0
	Bestseller WI	Periodic Update	-	SI-3
	Admin Confirm WI	Periodic Update	-	SI-3
SCADr	Users Followed	-	-	SI-0
	Recent Thoughts	-	-	SI-0
	Thoughtstream	Constraint on #subscriptions	-	SI-0
	Find User	-	-	SI-0

Table 4.1: The query modifications and indexes required for scale-independent execution of SCADr and TPC-W. Queries whose SQL is identical to that of another (e.g., Search By Subject WI and New Product WI are omitted in the interest of brevity.)

4.3.3.1 TPC-W

Although many changes were expected, in particular for the TPC-W queries, surprisingly few changes are required. Most notably, the TPC-W queries require rewriting more general `LIKE` predicates as tokenized keyword searches. This change is an artifact of the current implementation, as PIQL only supports inverted full-text indexes for such queries. The only real change required from the developer is the addition of a cardinality constraint on the number of items inside a shopping cart, though this limit is already defined as an optional constraint in the TPC-W specification. All TPC-W queries except the more analytic "Best Seller" and "Admin Confirm" are already scale-independent. In addition to the primary keys, the compiler automatically creates 5 indexes to support all queries more efficiently.

4.3.3.2 SCADr

The queries for SCADr require a limit on the number of possible subscriptions per user, similar to how Facebook limits the number of friends, as well as on the number of results shown per page. In the scale experiment, I set the limits to 10 subscriptions and 10 results per page. Refer to Section 4.4.4 for more detail on how different cardinality limits affect query performance.

4.3.4 Scale Experiments

In order to evaluate the effect of scale-independent optimization on actual performance at scale, both benchmarks are run on clusters of various sizes and the web interaction latency is recorded. For each data point, the amount of data per server is kept constant while the number of storage nodes and client libraries issuing queries increases. After the modifications from the previous section were made, PIQL's optimizer is able to find query and update plans that satisfy all of the scale-independent invariants.

4.3.4.1 TPC-W

TPC-W is scaled by first bulk loading 75 Emulated Browsers' worth of user data for each storage node in the cluster. The number of items is kept constant at 10,000 (as specified by the TPC-W spec), while user data is scaled linearly with the number of machines. Each piece of data is replicated on two servers both for availability and performance reasons. One client machine with the PIQL library is present for every two storage servers in the system The number of storage servers varies from 20 to 100, and this configuration results in clusters of up to 150 EC2 instances including clients. Each client executes the queries from the workflow specified by the TPC-W benchmark in 10 concurrent threads. Throughput and response time values are collected in 5-minute intervals, with at least 5 iterations for each configuration. The first run of any given setup is discarded to avoid performance anomalies caused by JITing and other warm-up effects.

Figure 4.3 shows that the response time per web interaction stayed virtually constant, even in the 99th percentile, independent of the scale. Thus, PIQL and its execution engine are able to preserve the scalability and predictable performance of the underlying key/value store even for a complicated application like TPC-W. Note, the response times from this experiment are not directly comparable with the predicted response times. This discrepancy is due to the fact that full web interactions also result in puts to the key/value store, and thus measured response time is slightly higher than predicted total response time.

Furthermore, Figure 4.4 demonstrates an additional benefit of scale-independent plan selection. Since the amount of work for each query is bounded, it is possible to obtain a near linear scale-up of throughput as the number of servers and clients increases.

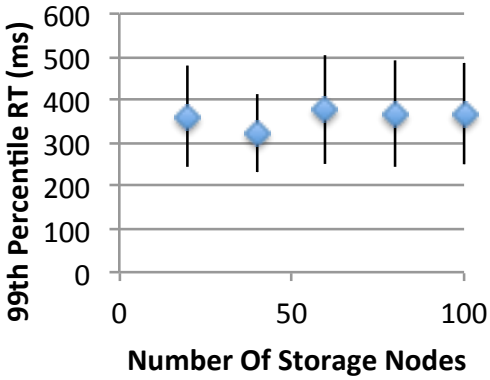

Figure 4.3: TPC-W 99th percentile response time, varying the number of servers

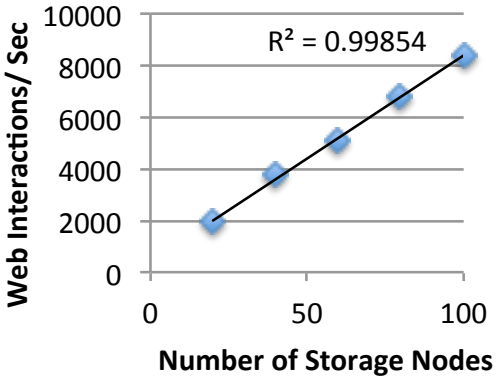

Figure 4.4: TPC-W throughput, varying the number of servers

4.3.4.2 SCADr

The SCADr benchmark is scaled using a methodology similar to the TPC-W benchmark by varying the number of storage nodes and clients. As with TPC-W, the data size increases linearly with the number of servers, with 60,000 users per server, 100 thoughts per user, and 10 random subscriptions per user. As with TPC-W, all data is replicated on two servers for increased availability. One client machine with the PIQL library is present for every two storage servers in the system. The number of storage servers is varied from 20 to 100 (including clients, this results in up to 150 EC2 instances).

Each client machine repeatedly simulates the rendering of the "home page" for SCADr by executing all of the given queries and measuring the overall response time. This execution and measurement are performed by 10 concurrent threads on each client machine. Throughput and response time statistics are collected in 5-minute intervals, with at least five iterations for each configuration. As in the previous experiment, I discarded the first run of any given

setup to avoid performance anomalies cause by JITing and combined all subsequent response time data to calculate a single 99th percentile value. Figure 4.5 and 4.6 show the results of this experiment. Again, the response time remains low and nearly constant as the system scales, even at the 99th percentile on a public cloud. Additionally, the second graph shows a near linear scale-up of throughput as the number of servers and clients increases.

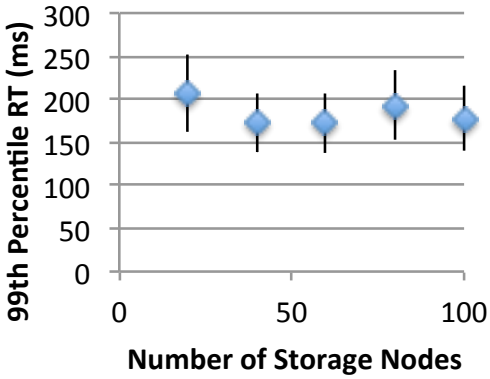

Figure 4.5: SCADr 99th percentile response time, varying the number of servers

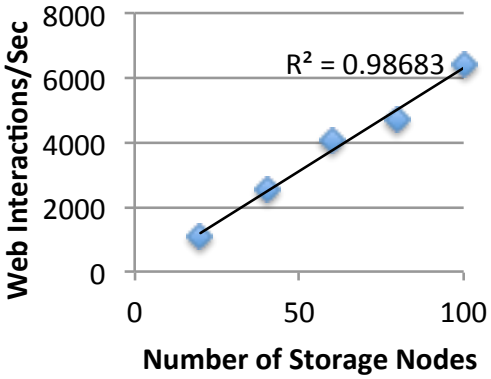

Figure 4.6: SCADr throughput, varying the number of servers

4.4 SLO Compliance Prediction

The previous sections described how the PIQL compiler converts a query into a scale-independent query plan that can execute under SI-0 or SI-1. If PIQL is unable to find such a plan, the query is reported to the developer as a possible performance risk at scale through the *Performance Insight Assistant* (see Section 4.4.4). Even if the compiler is able to find a bounded plan, however, it still does not guarantee that the plan is success-tolerant (i.e., that it can be executed in the targeted latency time frame). The number of tuples to process, although bounded, might still be too high to meet response-time objectives.

In this section, I describe the response time prediction model used by PIQL. This model calculates the risk that query operations will not complete in the time frame targeted by the applications SLO. Using this prediction model, the developer will be informed at compile time whether a bounded query is likely to meet its SLO. In the remainder of this section, I first describe the model of a single query plan operator and then how to compose operators together to evaluate the response time of the whole query as well as the risk of violating the SLO. Finally, I describe how the model is used as part of the Performance Insight Assistant.

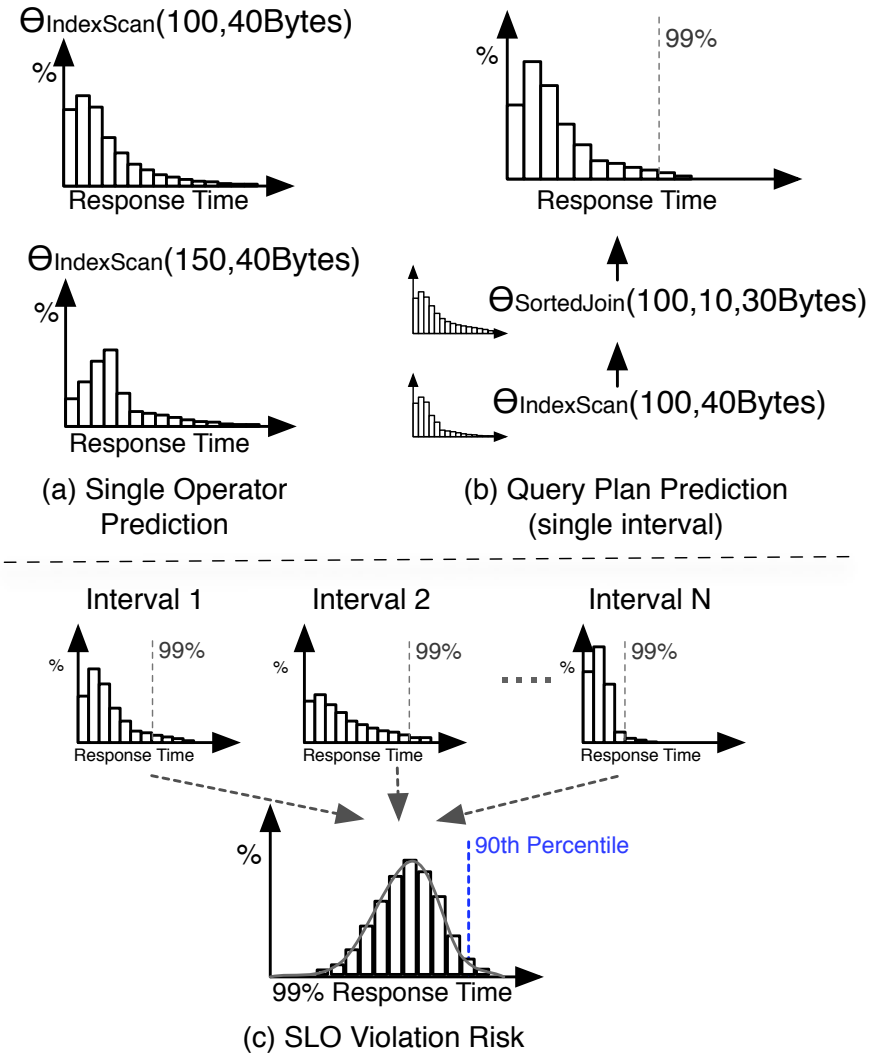

Figure 4.7: Modeling process for PIQL queries. The first step is to create models for each PIQL operator (a). Then, for each query, combine the operator models according to the query plan to create a PDF for the whole distribution (b). Finally, repeat the process of (b) for many timeframe histograms to better reflect the SLO response-time risk (c).

4.4.1 Single Operator Model

As described in the previous section, a physical query plan is composed of one or more operators. PIQL attempts to reason about the response time of the overall query starting with the expected performance of each operator. When performing this prediction, it is important to understand the environment where query execution will occur. PIQL is designed to run

on large clusters of commodity computers. While such an architecture has been proven to provide great computation ability at a reasonable cost, it can also result in highly variable performance [30]. To reflect the volatility in the response time, each operator is modeled as a random variable Θ.

To make model building tractable, the assumption is made that the response-time distribution of an operator only depends on the number of tuples and the maximum size of a tuple that it has to process. This simplification is reasonable, as the architecture is designed to avoid contention by automatically load-balancing and re-provisioning the key/value store as well as the application tier. The model is further constrained by only considering the three remote operators that interact with the key/value store. Empirically I found this simplified model to be sufficient, as I am primarily targeting interactive queries with SLO goals in the range of milliseconds up to a few seconds. Therefore, the latency to the key/value store is often the dominating factor.

However, network bandwidth and round-trip times are improving over time. Thus, in future versions it may be necessary to extend the model to include the local operators as well. Accordingly, the IndexScan operator can be modeled as $\Theta(\alpha, \beta)$, where α represents the maximum number of expected tuples (i.e., the bound enforced by a stop operator present in the logical plan) and β the size of each tuple. In contrast, the two join operators are described as $\Theta(\alpha_c, \alpha_j, \beta_j)$, where α_c represents the maximum number of tuples from the child operator (right relation), α_j the bound (e.g., schema cardinality) between the left and right relations, and β_j the maximum size of a tuple from the left relation. The model does not need to consider the size of the child tuples, as the transmission of these tuples over the network was already considered through the modeling of the operator that produced them.

An empirical distribution is obtained for each of the operator random variables and stored as a histogram. That is, as part of the model training, the response time behavior for every operator must be sampled by repeatedly executing the operator with varying cardinality and tuple sizes. This training is typically done once by setting up a production system in the cloud for a short period of time, where all operators are measured in parallel. This sampling collects a set of histograms for each of the three remote operators with different α and β values. Since these statistics are not application-specific, they can be pre-calculated for the most prominent public clouds (e.g., Amazon, Google, Microsoft). Figure 4.7(a) shows two possible distributions for the IndexScan operator with an expected cardinality limit α of 100 and 150 and a tuple size β of 40 Bytes. The models can be updated periodically as conditions in the datacenter change (e.g., as hardware is upgraded).

Given a query plan, the maximum cardinality α and the maximum tuple size β can be obtained for each operator from the optimzer annotations and the schema, respectively. Thus, choosing a distribution from the histogram collection becomes as simple as looking up the empirical measurements that correspond to the α and β values for a given operator. α is set to be the maximum cardinality to avoid underestimating the response time.

If the correct values are not present in the table, the (α, β) setting that is closest to the desired value while still being larger can be chosen instead. For example, consider the IndexScan given username on the *Subscriptions* table described in Section 1.6.1. It is known

from the schema annotation that the worst case cardinality is 150 and the tuple size is 40 Bytes. Hence, PIQL would choose $\Theta_{IndexScan}(150, 40\text{Bytes})$ from the two histograms in Figure 4.7(a).

It is often also possible to interpolate among the stored models to produce the desired model; this technique is suitable since the system exhibits a linear relationship between the cardinality and the response time. Since PIQL is designed for interactive applications, the response time goals for all queries are typically less than one second. For these purposes, reporting values with millisecond resolution is sufficient, so each histogram can be well-represented with on the order of one thousand bins. Therefore, while it is true that PIQL's modeling approach requires a separate histogram per (α, β) pair, this burden is not onerous. Due to the limited resolution of interest, each histogram can be stored in one kilobyte or two.

4.4.2 Query Plan Model

To predict the overall query response time, PIQL combines the operator models according to the physical query plan generated by the PIQL optimizer. Here, it is possible to make use of another property of the architecture. PIQL's execution engine is implemented as an iterator model and thus allows executing several operators in a pipelined fashion; however, since this thesis is focused on short-running queries, the latency can be modeled with sufficient accuracy assuming blocking operators. In the worst case, this simplification can result in the model failing to capture the overlap among the operators. Therefore, in some cases PIQL's prediction may be overly conservative. However, recall that PIQL's goal is not to predict response time but rather SLO compliance; thus, as long as the prediction is below the SLO, it still correctly predicts SLO compliance (see Section 4.5 for quantitative analysis).

Accordingly, the model can be simplified by assuming independence among the operators. For query plans (or plan sections) that are serial, the overall latency is represented by a random variable whose latency is the sum of the operator latencies, each of which is also represented by a random variable. For parallel plan sections, e.g. the two child plans of a union operator, one can determine the latency of each branch and then take the maximum. Since the latency of each operator is viewed as a random variable, summing the latency of two operators is equivalent to convolving their densities. Thus, to predict the latency distribution of a query, PIQL can simply convolve the densities of each operator. The resulting distribution is that of the query as a whole, as Figure 4.7(b) shows. Recall that the local operators are ignored by PIQL's response time modeling as requests to the key/value store dominate the latency. Accordingly, modeling the timeline query of SCADr shown in Figure 4.1 requires convolving two operators:

$$
\begin{aligned}
&Q_{ThoughtStream} = \\
&\quad \Theta_{IndexScan}(SubscrCard, SubscrSize) * \\
&\quad \Theta_{SortedJoin}(SubscrCard, ThoughtCard, ThoughtSize)
\end{aligned}
$$

4.4.3 Modeling the Volatility of the Cloud

The goal is to determine whether a query will meet its SLO regardless of the underlying database size. To do so, PIQL will inspect a predicted latency distribution for the query. PIQL's target developers are chiefly concerned with detecting violations of SLOs that are defined in terms of high quantiles of the query latency distribution. Thus, given an SLO such as "99% of queries during each ten-minute interval should complete in under 500 ms", if the 99th-percentile latency of the predicted distribution is less than 500 ms, then it can be predicted that the query will meet the SLO. Note that the length of the SLO interval affects its stringency: Longer intervals make the SLO easier to meet, as any brief periods of poor performance are counterbalanced by mostly good performance. In the following discussion, the assumption is made that the SLOs are defined over non-overlapping time intervals.

The 99th-percentile latency can vary from one interval to the next, which poses a further challenge for the model. As mentioned in Section 3.4, PIQL's ability to provide scale independence is based on the assumption that the key/value store's performance is relatively stable. Natural fluctuations in performance are particularly common in public clouds, where the machines and network are shared among many clients. The heavy workloads of some clients (e.g., Netflix's video encoding on Amazon) might cause short periods of poor performance, which could result in violations to an SLO even though it is routinely met under normal operation. Therefore, rather than providing a point estimate for the 99th-percentile latency of a given query, PIQL generates an estimation of its distribution, which captures how it varies from one interval to the next.

In order to estimate a distribution, PIQL takes the data collected from benchmarking the operators and bins the data according to the interval of interest; e.g., if the SLO is provided over a ten-minute interval, PIQL will create a separate histogram for each ten-minute period. This process allows PIQL to obtain a prediction of the query's 99th-percentile latency for each interval during which the benchmark was observed. Combining these predictions, as in Figure 4.7(c), results in a prediction of the distribution of the 99th-percentile latency. This distribution is a useful tool to a developer as it provides information about the risk of violating a query's SLO over time. For example, if the target response time equals the 90th percentile of the distribution, it means that for 10% of the intervals considered, the SLO goal may be violated.

4.4.4 Performance Insight Assistant

In order to make it easier for a developer to work within the constraints enforced by the PIQL optimizer, the system provides helpful feedback for fixing "unsafe" queries and for appropriately sizing cardinality limitations. Regarding the first case, any time a query is rejected by the optimizer, the developer is provided with a diagram of the logical query plan where the problematic segment is highlighted. The system provides the developer with possible attributes where the addition of a `CARDINALITY LIMIT` would allow optimization to proceed. For example, recall the thoughtstream query presented in Figure 4.1. If the

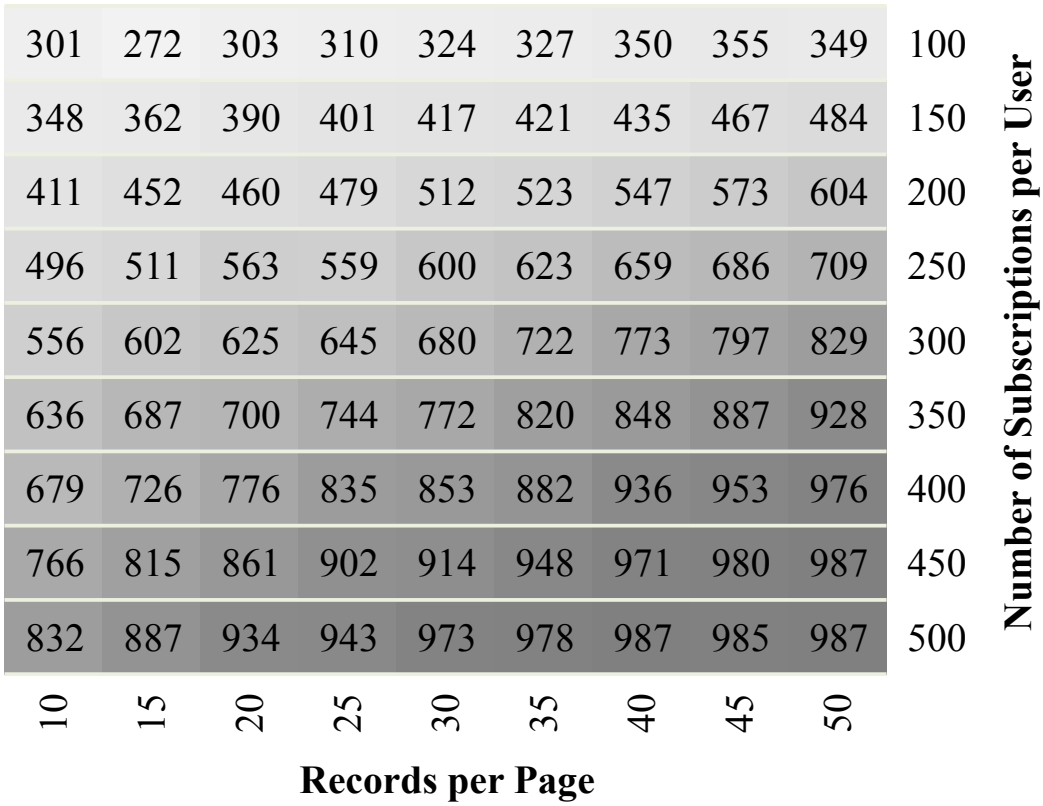

Figure 4.8: Predicted heatmap for 99th percentile latency (ms) for the thoughtstream query. On average, the predicted values are 13 ms higher than the actual values.

developer had not specified a limit on the number of subscriptions that a user could own, optimization would have failed. The assistant would have then pointed out that the problem was with the number of tuples produced by the subscription relation. The developer should then set a limit on the number of subscriptions per owner.

The Performance Insight Assistant also provides a sensitivity analysis of the run-time of a given query with respect to a developer's specified limit. This feature provides developers with guidance on how to set cardinality limits that are compatible with SLO compliance. Given a query written in PIQL, the system will provide the developer with a chart that shows how the 99th-percentile response time will vary for different cardinality limits. This chart can be generated by PIQL simply by predicting the latency distribution for each setting of the cardinality using the information from the per-operator benchmarks. Figure 4.8 shows this analysis performed for SCADr's thoughtstream query. The thoughtstream query has two parameters: The number of subscriptions a user has and the total number of thoughts to return. Since there are two parameters, choosing the cardinality limits for this query is more complicated, therefore a heatmap will be provided so that the developer can see what the latency would be for each combination of the parameters. The developer can

choose any of the cardinality pairs that would satisfy the query's SLO. Section 4.5 contains a complete query-by-query evaluation of the predication accuracy for a more comprehensive set of queries.

4.5 Evaluation of SLO Compliance Prediction

The accuracy of the SLO compliance prediction was evaluated emperically using the benchmarks first presented in Section 4.3. Table 4.2 shows the actual and predicted 99th-percentile values for each of the queries in TPC-W and SCADr. In this experiment, a constant cardinality for each query is maintained; for predictions with a varying cardinality, see Section 4.4.4. To train the prediction model, the operators are benchmarked on Amazon EC2 with a 10-node cluster using two-fold replication, and statistics were gathered for 35 10-minute intervals.

	Query	**Actual 99th (ms)**	**Predicted 99th (ms)**
TPC-W Benchmark	Home WI	94	95
	New Products WI	302	395
	Product Detail WI	118	125
	Search By Author WI	138	136
	Search By Title WI	122	145
	Order Display WI Get Customer	97	95
	Order Display WI Get Last Order	176	207
	Order Display WI Get OrderLines	126	138
	Buy Request WI	130	148
SCADr	Users Followed	113	141
	Recent Thoughts	88	89
	Thoughtstream	140	153
	Find User	84	82

Table 4.2: A comparison of the predicted and measured 99th percentile latency for each query in the TPC-W and SCADr benchmarks.

A prediction of the per-query 99th-percentile latency for each interval is obtained using the technique described in Section 4.4.3. In the table, for both the actual and predicted

cases, the max 99th-percentile value is reported. Taking the max corresponds to a very conservative approach to setting the cardinality. As the results show, the model slightly overestimates the actual 99th-percentile value in most cases. As I mentioned in Section 4.4, the goal of the PIQL system is to predict not response time but rather SLO compliance; thus, it is preferable to overestimate as long as the difference between the predicted and actual values is not so large as to be untrustworthy.

Three queries' response times are underestimated by the model by 2 ms, an insignificant error for the purpose of determining SLO compliance. PIQL's prediction model is most conservative for TPC-W's New Products WI. This overestimation is caused by pipelining that occurs between the query's two index foreign key joins, which the model does not currently capture. Future work may extend PIQL's model to handle this case.

If the SLO is sufficiently above the predicted value, the developer will still be able to make the correct decision regarding SLO compliance. However, a consequence of this modeling error is that the Performance Insight Assistant could potentially recommend a cardinality value lower than what the system could actually handle while still meeting its SLO. Thus, developers should take the assistant-recommended cardinality as a starting point and potentially increase the cardinality over time if the performance of the deployed application seems to consistently be well below the SLO. The SLOs provided with the TPC-W specification, ranging from 3 to 5 seconds for the queries evaluated here, are given in terms of the 90th-percentile end-to-end response time, whereas this evaluation analyzes the 99th-percentile query response time. Therefore, they are not directly comparable with these results. However, even at the 99th percentile, the running time of PIQL queries is significantly less that the given SLOs, implying that the queries are not the bottleneck to achieving the SLOs.

A target latency of 500 ms was chosen for the SCADr queries, since no SLOs exist for this benchmark as it was devised for the purpose of this thesis. This response time was chosen because longer server delays have been shown to affect the number of queries performed by a user [73]. The queries all complete within this bound even for the worst-case 99th-percentile response time.

4.6 Summary

By changing the objective function for query optimization to rule out plans that could possibly perform an unbounded number of operations, PIQL can bound the amount of work done by queries in an application. This section described PIQL's techniques for mapping logical query plans onto bounded sets of key/value store operations, thus maintaining Invariant 1. In doing so, the optimizer takes advantage of extra information in the query and the applications schema to bound both the final and intermediate results. I also showed that bounding the number of key/value store operations for a query allows the performance insight assistant to accurately reason about the worst case query performance and thus the applications SLO compliance. Together, these techniques let developers benefit from the high-level, declarative nature of SQL, while still providing a strict upper-bound on the latency for query execution.

Chapter 5

Precomputing Query Results

5.1 Introduction

The previous chapter showed how an optimizer could bound the number of operations performed by queries that only read data either from base relations or secondary indexes. However, as I explained in Chapter 1, not all queries can be answered on-demand in a scale-independent manner using only these data sources. Fortunately, in many cases, it is still possible to answer these queries if the results are incrementally computed each time data is added to the database. PIQL performs this precomputation through the creation of incrementally maintained materialized views (IMVs).

Anytime PIQL encounters a query where it is unable to find an execution plan in SI-0 or SI-1, the optimizer will check if the creation of an IMV could allow scale-independent execution. Figure 5.1 shows the four phases that are used to perform IMV selection in the context of PIQL's full optimization workflow. The first phase, view construction (Section 5.2), takes the PIQL query and transforms it into a view that can be used to answer the query for any value of the query's runtime parameters. However, even if this IMV enables the scale-independent execution of the original query, it is still important to analyze the IMV itself for scalability. Specifically, PIQL will ensure that bounds exist on both the cost of storing (Section 5.3) and maintaining (Section 5.4) the IMV before allowing its creation. Queries that are answered using a scale-independent view are executed under SI-2.

Since maintaining an even workload partitioning is key to PIQL's approach for ensuring predictable query performance, the optimizer must also analyse any created indexes to ensure they are not prone to insertion hotspots (Section 5.6). While it is clearly not possible to predict and avoid all possible hotspots a priori, PIQL provides developers with schema annotations that enable the detection of a particularly common type. When such a hotspot is encountered, phase four of view selection will rewrite the query to ensure that insertions will be evenly distributed across the cluster. However, since the data is now distributed across many machines, the answer to the original query must instead be computed periodically and cached. Due to the extra work required for this periodic update, queries that require this

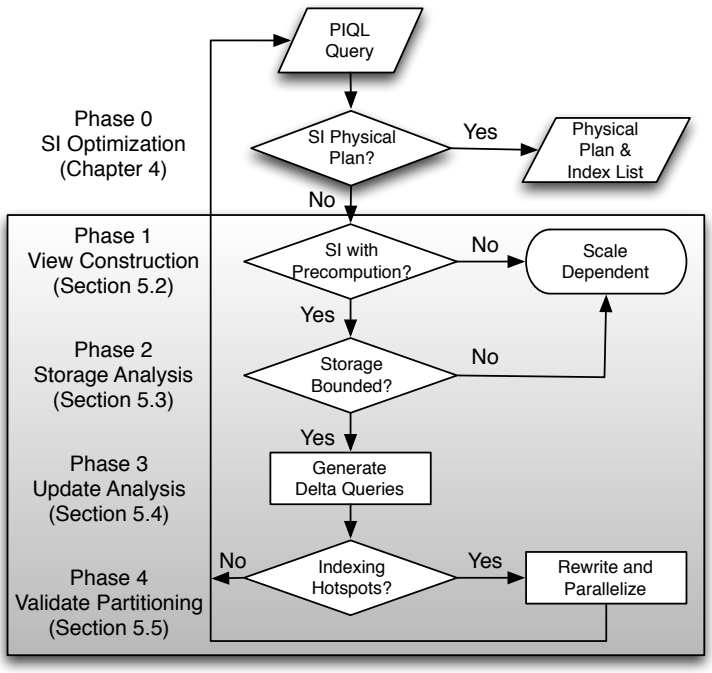

Figure 5.1: The phases of query optimization in the PIQL system. The shaded portion denotes the phases that are responsible for the selection of materialized views.

strategy are executed under SI-3.

5.2 View Construction

View construction, the first phase of PIQL's view selection system, is invoked when the optimizer is unable to find a scale independent physical plan for a given query. Since this query does not fall into either SI-0 or SI-1, the optimizer will instead try to answer the query by creating an IMV. Note that this is only the first step in scale-independent view selection and merely constructs a view that *could* be used to answer the query while satisfying Invariant 1. The algorithms in this section do not yet ensure that this view will meet the other invariants regarding storage and maintenance costs.

The general form of the IMVs created is as follows:

```
CREATE MATERIALIZED VIEW <viewName>
SELECT A_view [A_agg]
FROM r_1, r_2, ...r_n
WHERE P_view [GROUP BY A_eq]
```

A_{view} is an ordered list of the attributes projected by the IMV, while A_{agg} contains any aggregate expressions. The relations r_1 to r_n are the base relations present in the original query, and P_{view} is the set of predicates for the IMV. In the case of queries with aggregate expressions, a `GROUP BY` clause is also added to the IMV definition.

The construction algorithm ensures that the created IMV will allow scale-independent execution of the original query by making assumptions about the underlying storage structure. Specifically, this technique requires scale-independent access to tuples stored in the view given a prefix of the attributes present in A_{view}. Recall from Chapter 2 that examples of acceptable storage systems, given this constraint, include B-Trees and range-partitioned distributed key/value stores. Towards this end, I adopt the convention that the SQL expression used to define IMVs must specify not only which tuples are present in the view, but also the ordering of attributes in the clustered index used to store the view. Thus, the definitions of created IMVs effectively define the spatial locality of the precomputed tuples, allowing efficient retrieval of consecutive tuples.

5.2.1 View Construction Without Aggregates

The view construction algorithm presented in this section handles *select-project-join* queries with *conjunctive* predicates. Section 5.2.2 expands this algorithm to enable support for queries with aggregate expressions. At a high level, this algorithm is solving a variant of the view selection problem where the result of the target query must be computable by scanning a contiguous section of the view for any value of the runtime parameters. I start by describing how predicates that include runtime parameters are handled. Next, I describe the checks employed to ensure that the result of the original query can be obtained by scanning a contiguous section of the selected view. Finally, I describe the construction of the final view definition.

View construction starts by parsing predicates of the form *attr* `=` `<parameter>` or *attr* $\{<, >, \leq, \geq\}$ `<parameter>`. Using these predicates, the algorithm constructs the sets A_{eq} and A_{ineq}, which contain attributes occurring in equality and inequality predicates respectively. These attribute sets are added to the prefix of the projection of the view, allowing the predicates of the original query to be evaluated efficiently using an index scan.

As a concrete example, for the `twoTags` query (Section 3.3.1) the set A_{eq} contains {`t1.tag, t2.tag`}, and A_{ineq} is empty.

Next, Algorithm 3 checks to ensure that the answer to the original query for any set of runtime parameters will be a contiguous set of tuples in the view. The requirements for this condition are as follows: Any number of attributes may be filtered by equality with a runtime parameter, but the query may contain at most one predicate where an attribute is filtered by an inequality with a parameter. This limitation is due to the fact that computing the intersection of the two inequality predicates may involve scanning over an arbitrary number of tuples in the view, and thus could violate Invariant 1. Similarly, any number of attributes may be specified in the `ORDER BY` clause, but this ordering must be prefixed by the inequality attribute, if one exists.

Algorithm 3 Confirming Adjacency of Result Tuples

1: A_{order} := ordered list of attributes in `ORDER BY` clause
2: **if** $|A_{ineq}| > 1 \vee (|A_{order}| > 0 \wedge A_{order}[0] \neq A_{ineq})$ **then**
3: **return** $false$
4: **end if**
5: **return** $true$

If Algorithm 3 returns successfully, it then creates the set P_{view}. This step is accomplished by removing any predicates that involve a runtime parameter and then simplifying any redundant equality predicates.

To avoid changing the meaning of the query, this procedure must not inadvertently discard any transitive equality constraints present due to parameters that appear multiple times in the query. For example, given a query with the predicates a_1 `= <p1>` and a_2 `= <p1>`, P_{view} must contain the predicate $a_1 = a_2$. These transitive equalities are accounted for by generating predicates for the view from the equivalence classes defined by the equality predicates in the original query. Inequality predicates, in contrast, are copied directly from the original query. These predicates will eventually be turned into inequalities with parameters during delta query calculation (Section 5.4). Therefore, the rules regarding multiple attributes participating in inequalities still apply, and thus the creation of the view could be later be rejected due to a lack of a scale-independent maintenance strategy.

Algorithm 4 describes this process of creating P_{view}. Taking as input the set of conjunctive predicates (Line 1), the algorithm starts by partitioning values found in P into equivalence classes (Line 2). Next, for every equivalence class (Line 3), the algorithm adds an equality predicate to P_{view} for each attribute pair in the class (Line 4-11). Finally, the algorithm copies inequality predicates involving non-parameters into P_{view} (Line 15-21).

As an example, consider again the `twoTags` query. The predicates in this query define three equivalence classes:

```
{t1.docId, t2.docId, d.id}
{t1.tag, <tag1>}
{t2.tag, <tag2>}
```

From these classes the following is produced for P_{view}:

```
{t1.docId = t2.docId, t2.docId = d.docId}
```

Once P_{view} has been constructed, the system next creates A_{view}. A_{view} contains the attributes in A_{eq} and A_{ineq} as well as any remaining key attributes from the top-level tables present in the original query. These key attributes are added to the projection in order to ensure that the view can be efficiently maintained using production rules (Section 2.3.2), as proposed by Ceri and Widom [21]. Specifically, the view selector ensures that there will be no duplicate tuples in the view and that all top-level table references are safe. Additionally, to

Algorithm 4 Generating View Predicates

```
1: P := set of all conjunctive predicates in the query of the form v1 op v2
2: EquivalenceClasses(P) := set of equivalence classes under P
3: P_view ← {}
4: for all X ∈ EquivalenceClasses(P) do
5:     prevAttr := ∅
6:     for all v ∈ X do
7:         if !isParam(v) then
8:             if prevAttr ≠ ∅ then
9:                 P_view := P_view + Equality(prevAttr, v)
10:            end if
11:            prevAttr := v
12:        end if
13:    end for
14: end for
15: for all p ∈ P do
16:    if isInequality(p.op) then
17:        if !isParam(p.v1)∧!isParam(p.v2) then
18:            P_view := P_view + p
19:        end if
20:    end if
21: end for
```

avoid unnecessary redundancy, any key attributes that are unified by an equality predicate in the original query will only appear once in the projection of the view.

Algorithm 5 describes the process used to populate A_{view} and starts by initializing A_{keys} to be the set of key attributes for all top-level tables from the original query (Line 2). Next, it initialises A_{view}, the ordered list of attributes that will appear in the view, and $A_{covered}$, the set of attributes already represented in the view definition considering unification, to be empty (Lines 3-4). Then, it iterates over the ordered concatenation of the attribute sets (Line 8) and adds to the view any attributes not yet present in the cover set $A_{covered}$ (Line 10). To prevent redundant values from appearing in the view, when an attribute is added, its entire equivalence class is added to $A_{covered}$ (Line 11).

Applying Algorithm 5 to the `twoTags` query, the attributes `t1.tag`, `t2.tag`, `d.timestamp`, and `d.docId` are selected. The first two attributes come from A_{eq}, `d.timestamp` comes from A_{order}, and `d.docId` from A_{keys}. By placing attributes from A_{eq} first, the view selector ensures that tuples satisfying the original query can be located by a prefix of the keys in the view. Including `d.timestamp` next ensures that the relevant tuples in the view will be sorted as specified by the `ORDER BY` clause. Finally, `d.id` allows for safe view maintenance and the retrieval of the actual document.

Continuing the `twoTags` example, the following materialized view is created by the view

Algorithm 5 Choosing View Keys

1: $R :=$ the set of relations present in the original query
2: $A_{keys} := \{a : r \in R, a \in keyAttrs(r)\}$
3: $A_{view} \leftarrow []$
4: $A_{covered} \leftarrow \{\}$
5: **function** EQUIVALENCECLASS(P, a)
6: return all values in P unified with a due to equality predicates, including a itself
7: **end function**
8: **for all** $a \in A_{eq} + A_{ineq} + A_{order} + A_{keys}$ **do**
9: **if** $a \notin A_{covered}$ **then**
10: $A_{view} := A_{view} + a$
11: $A_{covered} := A_{covered} \cup EquivalenceClass(P, a)$
12: **end if**
13: **end for**

constructor:

```
CREATE MATERIALIZED VIEW twoTagsView
SELECT t1.tag as t1tag, t2.tag as t2tag,
      d.timestamp, d.docId
FROM Tags t1, Tags t2, Documents d
WHERE t1.docId = t2.docId AND
     t1.docId = d.docId
```

Once the view has been constructed for a given query, the query is rewritten by replacing the top-level tables with the view and renaming any attributes to their equivalent attribute in the view. If there are any attributes that are present in the original query, but not in the view, they can be retrieved either by joining the view with the base relation on the keys that are present or by adding the missing attributes to the view definition. The former will require more computation at query time, while the latter will require more storage for the view. Since the choice of method does not affect the scale independence of the query, the system decides which technique to use based on the predicted SLO compliance of the resulting query.

This final step rewrites the `twoTags` query to use the materialized view as follows:

```
SELECT t1.docId
FROM twoTagView
WHERE t1tag = <tag1> AND
      t2tag = <tag2>
ORDER BY timestamp
LIMIT 5
```

5.2.2 View Construction With Aggregates

PIQL's view selection system is also capable of handling many queries that contain aggregates in the target list. In this subsection, I describe both the class of aggregates that are supported and the alternative view selection algorithm used when an aggregate is present in a query.

5.2.2.1 Scale-Independent Aggregates

The class of scale-independent aggregates is defined in part by the storage requirements for partial aggregate values. Using the categories of aggregates first defined by Gray et al. [38], PIQL can safely store partial aggregate values for both *distributive* aggregates (such as `COUNT` or `SUM`) and *algebraic* aggregates (such as `AVERAGE` and `VARIANCE`). Both of these categories have partial state records of fixed size. In contrast, *holistic* aggregates such as `MEDIAN` can require an unbounded amount of partial state to be stored and thus conflict with the goal of scale independence.

Bounding the storage required for each partial aggregate value alone is not sufficient, as a scale independent system must also ensure that efficient incremental maintenance of aggregated values is possible. Towards this end, PIQL also requires that updates to the aggregate be both associative and commutative. While both `MIN` and `MAX` are distributive aggregates, updates to them do not always commute in the presence of deletions. For example, when the maximum value from a given group is deleted, the only way to update the aggregate is to scan over an unbounded set of tuples looking for the new maximum value.

5.2.2.2 View Selection with Aggregates

When PIQL's view selection system detects an aggregate in the projection of a scale-dependent query, it uses a slightly modified view construction algorithm. I explain these modifications in four parts.

Ordering Views containing aggregates cannot have any inequalities with parameters, as it could require unbounded computation in order to maintain such views. Additionally, since each query containing an aggregate will return only one tuple, there is no `ORDER BY` clause. These two changes eliminate the need for Algorithm 3.

Keys Views containing aggregates are maintained using techniques analogous to a counting solution [40]. Thus, it is not necessary to use Algorithm 5 to add keys for safe maintenance.

Aggregate Expressions The view selection algorithm must add partial aggregate values A_{agg} to the created view. In the case of distributive aggregates, only the aggregate expression itself must be added, while for algebraic aggregates more information may be required to ensure efficient incremental maintainability. For example, `AVERAGE` is computed by keeping both a `SUM` and a `COUNT`.

Group By Clause All of the attributes in A_{eq} are added to the `GROUP BY` clause of the created view.

The following is an example of how this modified algorithm would select a view for the `countTags` query, which calculates the number of documents assigned a given tag.

```
SELECT COUNT(*)                CREATE MATERIALIZED VIEW docsPerTag
FROM Tags                      SELECT tag, COUNT(*) as count
WHERE tag = <tag>              FROM Tags t
                               GROUP BY tag
```

5.2.3 Views for Window Queries

PIQL can handle many queries that operate over windows of data, and the techniques for handling these queries fall into two categories. For queries that operate over a fixed size tuple window, an index can be created on insertion timestamp. Since the tuple window bounds the number of tuples that will need to be retrieved from this index, scale independent optimization can then be performed on the rest of the query using standard techniques. Section 5.6 discusses the special consideration required when creating such indexes to avoid hotspots as the size of the database scales.

Aggregate queries that operate over a time window are handled by prepending an *epoch* identifier to the beginning of the view. The epoch identifier calculated using the following formula: $timestamp - (timestamp \mod windowSize)$. For example, consider modifying the `countTags` query from the previous section to count tags for a sliding window of length one minute. PIQL would create the following view (assuming timestamp is measured in milliseconds).

```
CREATE VIEW docsPerTagWindowed
SELECT (timestamp - (timestamp % (60*1000)),
        tag, COUNT(*) as count
FROM Tags t
GROUP BY (timestamp - (timestamp % (60*1000)), tag
```

Queries where the window size is larger than the slide amount can also be handled but will result in updates to all relevant epochs. Stale epochs can be garbage collected.

5.3 Bounding Storage Costs

Once a candidate view has been produced by the selection algorithm described above, Phase 2 performs a static analysis of the maximum possible storage requirements. If the analysis

determines that the view could grow super-linearly relative to the size of the base relations, the view is rejected, as its creation might violate Invariant 3.

PIQL's view size analysis utilizes dependency information from both the schema and the view definition. Note that the dependencies defined in this section subsume standard functional dependencies, where the latter can be represented as a cardinality dependency of weight one.

At a high level, the algorithm bounds the maximum size of view by determining how many degrees of freedom remain after taking into account all of the tuples from a single relation in the view definition. In doing so, the algorithm determines whether the dependencies present are sufficient to ensure that the view is bounded in size by a constant factor relative to at least one base relation.

5.3.1 Enumerating Dependencies

The analysis starts by constructing the list of all dependencies for a given view definition. These dependencies are generated by analyzing the view as well as the schema of the application using the following four rules:

1. $(keyAttributes) \rightarrow (otherAttributes)$
   Add a dependency of weight one to represent the functional dependency due to the primary key's uniqueness constraint.

2. $(keyAttributes) \xrightarrow{cardinality} (constrainedFields)$
   Add a dependency for each relation that has a cardinality constraint declared in the schema, weighted by the cardinality of the constraint.

3. $attribute_1 \leftrightarrow attribute_2$
   Add a bidirectional dependency of weight one for each attribute pair present in an equality predicate in the `WHERE` clause of the view definition.

4. Fixed Value $\rightarrow attr$
   Add a directional edge from a special fixed node to any attribute that is fixed by an equality predicate with a literal (e.g. $attr$ = true). This dependency is always implied, independent of what other dependencies are being considered.

Take, for example, the view definition of the `twoTags` query from the previous section. Given this view definition, the algorithm will produce the following list of dependencies. I omit trivial dependencies for the sake of brevity.

$$d.id \rightarrow (d.timestamp, \ldots) \tag{5.1}$$

$$t1.docId \xrightarrow{K} t1.tag \tag{5.2}$$

$$t2.docId \xrightarrow{K} t2.tag \tag{5.3}$$

$$t1.docId \leftrightarrow t2.docId \tag{5.4}$$

$$t1.docId \leftrightarrow d.docId \tag{5.5}$$

First, dependency 1 is added to the set due to the primary key of the document relations. Next, dependencies 2 and 3 are added to the set due to the cardinality constraint that each document may have no more than K tags. Finally, dependencies 4 and 5 are added as a result of the equality predicates present in the view definition.

5.3.2 Bounding Size Relative to a Relation

The following algorithm determines if the maximum size of the view is linearly proportional to one of the relations present in the query, using the dependency list generated by the above rules. This analysis is performed by finding a relation present in the query such that all attributes present in the view are functionally dependent on the primary key of the selected relation. If independent attributes remain after the inclusion of all possible dependencies, then a bound on the size of the IMV does not exist relative to that relation. If no relation can be found such that all attributes are functionally dependent on its primary key, then the view is rejected due to a possible violation of Invariant 3.

Algorithm 6 Bounding Maximum View Size

```
1:  R := the set of all relations present in the view
2:  D := the set of dependencies for the IMV
3:  for all r ∈ R do
4:      A_dep := keyAttrs(r)
5:      repeat
6:          A := {a | a ∈ attrs(R), a ∉ a_dep, D ⊨ A_dep → a}
7:          A_dep := A_dep ∪ A
8:          if A_dep ⊇ attrs(R) then
9:              return true
10:         end if
11:     until |A| = 0
12: end for
13: return false
```

Algorithm 6 describes this process formally and takes as input the set of all relations present in the view and all dependencies for the IMV (Lines 1-2). It returns a boolean value

indicating if there is an upper bound on the size of the view due to these dependencies. The algorithm iterates over all of the relations present in the view definition. For each relation, the set of attributes functionally dependent on this relation a_{dep} is initialized to the key attributes of the relation (Line 4). Then the algorithm iteratively selects the set of attributes a that are functionally dependent on the attributes in a_{dep} given D but are not yet present in a_{dep} (Line 6). These new attributes are then added to a_{dep} (Line 7). If a_{dep} now includes all attributes from the view, the optimizer knows the view size is bounded relative to the relation r, and the algorithm returns $true$ (Lines 8-10). The iteration stops if at any point there are no more attributes to add to the set (Line 11). If no bound can be found for any relation, the algorithm returns $false$ (Line 13).

To understand this process more concretely, consider again the definition of the IMV created by the PIQL system to answer the `twoTags` query. Algorithm 6 will start by selecting the Tags (t1) relation. This initializes a_{dep} to {`t1.id, t1.tag`}. On the first iteration, it will add {`t2.docId, d.docId`} to a_{dep} due to dependencies 4 and 5, respectively. Then, on the second iteration, it will add all remaining attributes due to dependencies 1 and 3. At this point, since all attributes in the query are in a_{dep}, the algorithm will return $true$.

Note that Algorithm 6 would return $false$ were it not for the cardinality constraint on the number of tags per document (functional dependencies 2 and 3). To understand how this schema modification could cause the `twoTagView` definition to violate Invariant 3, consider the degenerate case of a database with only a single document. As the number of tags increases, the size of the view would grow quadratically.

5.3.3 Views with GROUP BY

When the view contains a `GROUP BY` clause, the system must modify the procedure for determining if the storage required by the view is bounded. This modification is a result of the fact that the `GROUP BY` effectively collapses many tuples down to those with unique values for the attributes in the `GROUP BY`. Thus, instead of requiring attributes from all relations to be covered by the dependencies, it is sufficient to have dependencies that cover only the attributes being grouped on. This change can be implemented simply by substituting $attrs(R)$ with the set of attributes present in the `GROUP BY` on Line 8 of Algorithm 6.

5.4 Bounding Maintenance Cost

Once the view selection system has produced a candidate view and verified that a bound exists for the storage required by the view, Phase 3 ensures the existence of an upper bound on the number of operations required by incremental maintenance, given a single update to any of the relations present in the view. In this section, I first review the standard techniques used to perform incremental maintenance. I then explain the analysis performed to ensure that the total number of operations required by this mechanism will be bounded.

5.4.1 Maintenance Using Production Rules

Recall from Section 2.3 that incremental maintenance can be performed through the use of production rules [21] that execute each time a base relation is modified. At a high level, the production rules update the view by running a *delta query*. This delta query calculates all of the tuples that should be added or removed from the view due to a single tuple insertion or deletion. An updated tuple is processed as a delete followed by an insert.

Since PIQL's view construction algorithm ensures the safety of incremental maintenance, the delta query for an update can be derived by substituting the updated relation with the single tuple being inserted or deleted. In order to understand the delta query's derivation more concretely, consider the IMV `twoTagsView` when a new tag is inserted. PIQL's view selection system would calculate the first delta query by substituting the relation `t1` with the modified tuple in the view definition. Notationally, I represent the values of inserted or deleted tuple as `<parameters>` to the delta query. This substitution produces the following rule which will be run anytime a tuple is added to the `Tags` relation:

```
CREATE RULE newTag ON INSERT Tags
INSERT INTO twoTagsView
SELECT <tag>, t2.tag as t2tag, d.timestamp, d.docId
FROM Tags t2,
     Documents d
WHERE t2.docId = <docId> AND
      d.docId = <docId>
```

Rules containing delta queries must be created for each of the relations present in the view definition, though I omit the remainder of this process for brevity. It should be noted that for queries containing self-joins, all delta queries for the modified relation must be run for an inserted or deleted tuple. For example, when a new tag is inserted the delta query for both `t1` and `t2` must be run to update the `twoTagsView`.

5.4.2 Maintenance Cost Analysis

Given all of the delta queries for a view, PIQL must verify that none of them threaten the scalability of the application by requiring an unbounded number of operations during execution. Fortunately, since these delta queries are represented as SQL queries, the view selection system can reuse the optimization techniques from Chapter 4 to perform this analysis. If the optimizer is able to find scale-independent plans for all of the delta queries, then the system can certify that the addition of the view will not cause a violation of Invariant 2.

For example, consider the delta query `newTag` from the previous subsection. Due to the cardinality constraint on the number of tags per document, this query can always be executed by performing a single sequential scan over a secondary index.

Since verifying the scalability of the delta queries involves invoking the optimizer again, it is possible that the only scale-independent physical plan for a delta query will also require the

creation of an index or materialized view. Thus, in some cases the creation of a materialized view could result in multiple *recursive* invocations of the view selection system. Fortunately, it is known that this recursion will always terminate since the degree of the query will decrease with each successive derivation [50].

5.4.3 Updating Aggregates

Materialized views that contain aggregates are also maintained using delta queries with one important distinction. Delta queries for aggregate functions return a list of updates to possibly existing rows instead of tuples that will be added or removed from the IMV. Notationally, I represent the update that will be applied to a given field in the view as an expression in the `SELECT` clause prefixed by a `+`.

The delta queries themselves are derived using rules similar to those used for other queries. Specifically, the inserted tuple is substituted for the relation being updated and simplified, leveraging the distributivity properties of joins and aggregates [50]. For example, considering the view for the `countTags` query, the following delta query is used for maintenance:

```
CREATE RULE ON INSERT INTO Tags
ON INSERT INTO Tags
UPDATE twoTagsCount
SELECT <tag>, t2.tag, +1
FROM Tags t2
WHERE t2.docId = <docId>
```

When a new tag is inserted for a document, this query increments the count of all combinations of this tag and others already present for that document. The rules for deletions are symmetric and are omitted for brevity.

5.5 Evaluation of Incremental Precomputation

Incremental precomputation's ability to convert previously problematic queries into scale-independent ones is evaluated empirically by running the `twoTags` query on various data sizes using both execution strategies. This query is interesting because it has no scale-independent physical plans in SI-0 or SI-1. Thus, without an IMV, the `twoTags` query can often be expected to return quickly, but common data patterns can result in arbitrarily slow response times. For example, consider a case where a large number of documents are assigned `tag1` but few documents are also assigned `tag2`. With only an index, the system will need to read many of the documents with `tag1` during query execution, resulting in query latency that grows proportionally with the number of tuples touched.

To validate whether this problematic scenario arises in practice, a dataset is constructed at varying scales. The cardinality for each document's tags is sampled from a Zipf distribu-

tion (n=2000, s=0.1). This distribution was chosen as it approximates the frequency of tags in a social context [29, 61].

The performance of the `twoTags` query is measured at scale by partitioning the `Tags` relation as well as its materialized view evenly across the cluster. Each partition is replicated twice for both availability and performance. For each data point, the experiment begins by bulk loading a set of initial tags into each partition, ensuring the number of tags chosen fits in the memory of the machine. This in-memory constraint was met by fixing the number of tags at 200,000 for each physical partition. This configuration guarantees that the amount of data and the number of machines increase at the same rate. A cardinality constraint of 10 tags per document is enforced, with an average of four tags per document initially. For each data point, the results from multiple runs across different EC2 cluster instantiations are combined, and the first run of any setup is discarded to mitigate JIT warm-up effects. A 2:1 ratio of storage to client nodes is maintained across cluster sizes, with each client utilizing two reader and two writer threads to issue requests to storage concurrently. Since writes are significantly more expensive than reads, this results in a read-write ratio to the database of approximately 25 to 1.

5.5.1 On-Demand vs. Materialization

The evaluation begins with a comparison of the execution time both execution strategies. Figure 5.2 shows that if the system attempts to execute the query using an unbounded query plan, the 99th percentile latency grows linearly with the size of the data, and the query read latency quickly rises to over a second. The significant increase in response time clearly demonstrates the danger of allowing queries with scale-dependent plans to run in production. In contrast, when the same query is executed under SI-2 with an IMV, the response time remains nearly constant, leveling off at a 99th percentile latency of 8ms.

Additionally, Figure 5.3 shows that the throughput of the application increases nearly linearly with the number of machines utilized when taking advantage of the IMV. In contrast, the performance suffers when attempting to run the same query using the unbounded physical plan due to the increasing cost of execution.

5.5.2 Cost of Incremental View Maintenance

The previous section demonstrated that the use of unbounded query plan can have a significant effect on the response time of queries in an application. Fortunately, when the `twoTags` query is instead executed using a physical plan from SI-2 the response time remains nearly constant at 8ms. However, since executing the query under SI-2 involves shifting some of the computational work to insertion time, the analysis must also verify that this extra work does not negatively impact the scalability of writes to the application. This verification is performed by measuring the write performance as both the size of the data and the number of machines in the cluster increase.

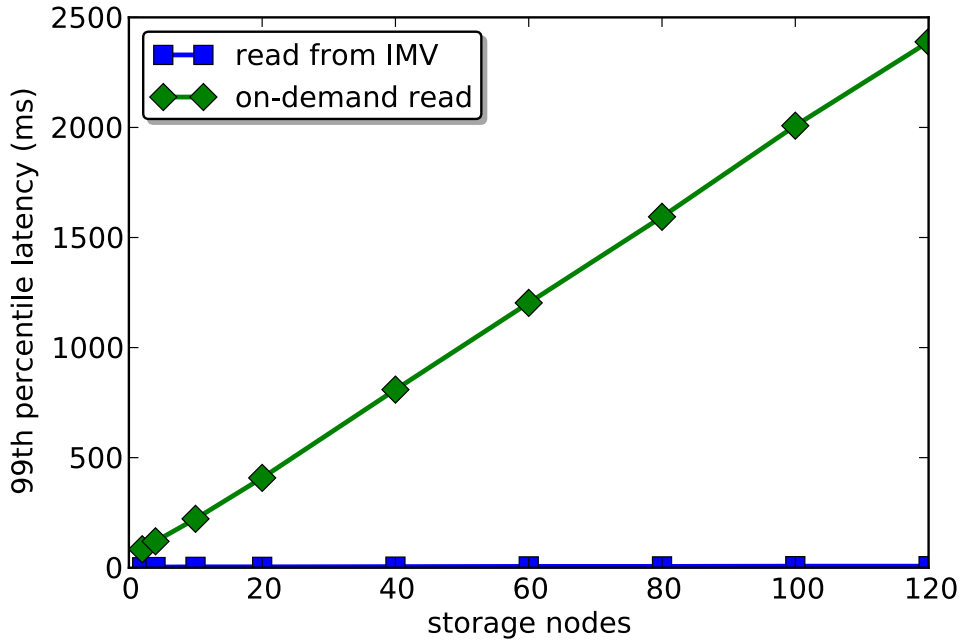

Figure 5.2: 99th-percentile query latency as cluster size grows. When executing the query with an unbounded physical plan, the latency of the twoTags query grows linearly with the scale of the application. In contrast, read latency is nearly constant (and much lower) when using a scale-independent query plan under execution level SI-2.

While the incremental view maintenance could be performed asynchronously, returning to the user immediately after the write is received, the system is instead modified to perform all maintenance synchronously for the purpose of this experiment. Figure 5.4 shows the effect that the maintenance of the materialized view has on write latency.

While latency of updates to the `Tags` relation is affected by view maintenance, the increase is relatively small ($\sim$ 110ms in the worst case) and remained virtually constant for cluster sizes larger than 40 nodes. The initial increase in write latency as the cluster grows is due to the fact that the number of partitions in the system was small with respect to the cardinality constraint on tags. This relatively large cardinality constraint means that, for a smaller cluster, fewer partitions are often contacted per write, since some writes will go to the same machine. However, since Invariant 2 bounds the number of writes that will be performed in the worst case, eventually the maximum distribution is reached and the performance effect leveled off. Developers who wish to reason about the worst case performance as the cluster grows could simply disable the batching optimization performed by the execution engine. Doing so would simulate the performance that could be expected when maximum possible distribution is achieved.

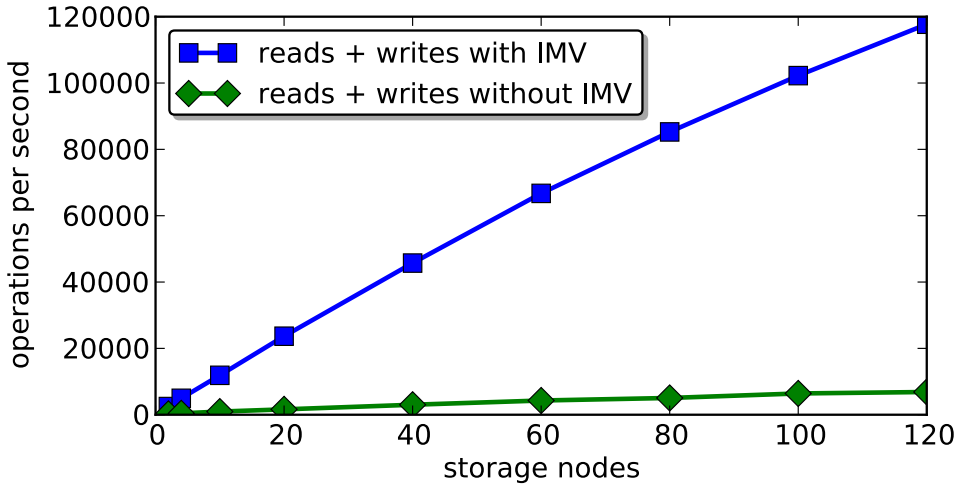

Figure 5.3: The throughput of the system increases linearly with the resources provided when using an IMV.

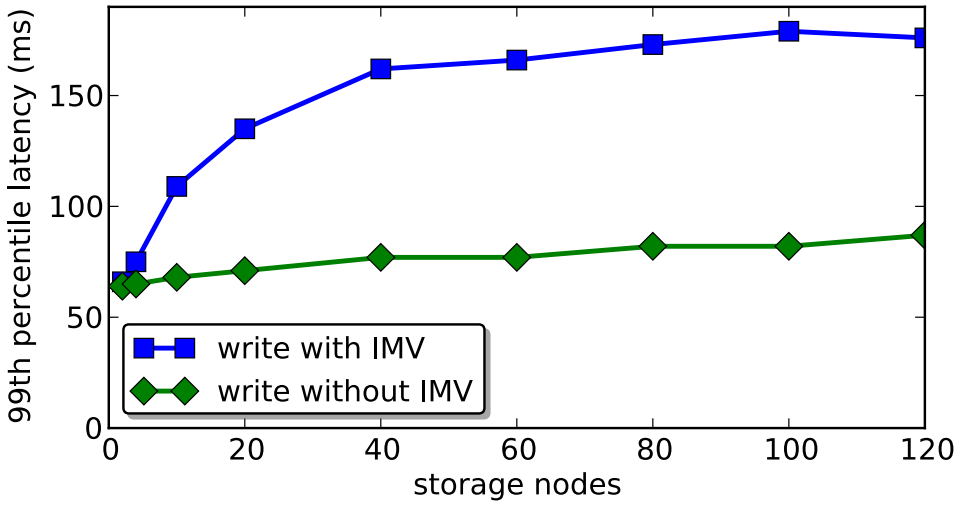

Figure 5.4: The write completion time remains bounded even with the additional overhead of incremental view maintenance.

5.6 Avoiding Common Hotspots

Preventing workload hotspots is critical when attempting to maintain consistent performance in a distributed query execution system. While clearly it is not possible to predict and avoid all possible hotspots, PIQL is able to detect some common cases using schema annotations provided by the developer.

Based on these annotations, PIQL will avoid creating indexes that could result in a

hotspot, and will instead suggest views that spread the insertions across the cluster. In this section, I describe an addition to standard SQL Data Definition Language (DDL) and the technique used to rewrite potentially problematic queries.

5.6.1 DDL Annotations

PIQL allows developers to annotate columns whose values exhibit strong temporal locality with respect to insertions. Figure 5.5 shows the canonical example of such locality, an index over the creation timestamp for a given record. Since all records created within a short time period will have similar values for this attribute, all updates to an index ordered over this attribute will be routed to the same partition. While this concentration of updates will not result in performance issues at a small scale, it is in direct conflict of the goal of predictable performance as the system grows.

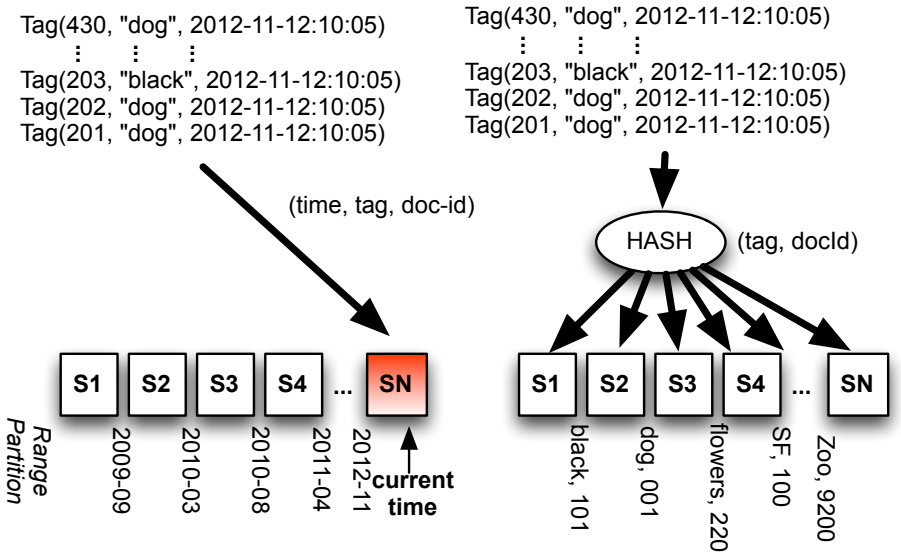

Figure 5.5: Indexes over timestamp can result in hotspots, denoted by the shaded server. In contrast, PIQL chooses to distribute insertions over all machines in the cluster.

Developers can warn the optimizer to avoid the creation of these hotspot-prone indexes by using the `TEMPORAL` keyword. For example, consider the following modification to the

Tags schema, first introduced in Section 2.3.2.

```
CREATE TABLE Tags (
  docId INT,
  tag VARCHAR,
  timestamp DATETIME TEMPORAL,
  CARDINALITY LIMIT 5 (docID)
);
```

5.6.2 Query Rewriting

When PIQL detects a hot-spot prone index, it instead creates materialized views that can be used to answer the query using a two step process. As an example of a case where this rewriting would occur, consider the following query, popularTags, which returns the most popular tags out of the most recent 5000 insertions.

```
SELECT tag, COUNT(*)
FROM (SELECT * FROM Tags
      ORDER BY timestamp
LIMIT 5000)
GROUP BY tag
```

If the developer failed to include the TEMPORAL keyword on timestamp, the optimizer would attempt to execute this query on-demand under SI-1 using an index over Tags.timestamp. This execution plan would be problematic at scale, however, as maintaining this index in a range-partitioned system would eventually lead to an overloaded partition and subsequently high query latency.

Instead of creating this hot-spot prone index, PIQL will instead create a view that is hash partitioned by the primary key of the relation. In the example below, this partitioning can be seen though the HASH keyword at the beginning of the view definition. Within each partition the records are sorted by the TEMPORAL attribute (i.e., timestamp in this example). Any remaining required attributes are appended to the end of SELECT clause. This procedure results in the following materialized view for the popularTags query.

```
CREATE VIEW popularTagStaging
SELECT HASH(tag, docId), timestamp, tag
FROM Tags
```

Next, the original query is rewritten as a periodically updated materialized view by substituting the original relation with the hash-partitioned materialized view. For the popularTags query this transformation results in the following SQL.

```
CREATE VIEW popularTags PERIODIC 1 MIN
```

```
SELECT tag, COUNT(*)
FROM (SELECT * FROM popularTagStaging
      ORDER BY timestamp
        LIMIT 5000)
GROUP BY tag
```

The optimizer will choose a physical plan that executes the subquery in parallel on each partition. Since the tuples in each of the partitions are sorted by the desired ordering attribute, each partition will only need to scan over 5000 tuples in the worst case. Therefore, the total amount of work that needs to be performed serially has a constant upper bound. However, since the total amount of computation is now proportional to the number of machines in the cluster, this query now falls into SI-3, unlike the twoTags query from the previous sections, which falls into SI-2. The next section demonstrates empirically that even though the total amount of work grows with the size of the cluster, the increased parallelism allows us to execute the query with only minor increases in the periodic update latency.

5.7 Evaluation of Periodic Refresh

The performance of periodically refreshed views is evaluated by executing the TPC-W benchmark, which was introduced in Section 4.3. Recall from earlier that two of the web interactions have no scale independent plans under SI-0 or SI-1. Fortunately, PIQL's automatic creation of materialized views allows for full scale-independent execution of the benchmark, including the analytic queries.

These two web interactions require the creation of materialized views due to the workload hotspots that would result from their unmodified execution. The first, the BestSellerWI, returns the top 50 most popular items from the last 3333 orders. The second, AdminConfirmWI, returns the 5 most common items co-purchased with a specified item from the last 10,000 orders. Since executing either of these interactions on-demand would require an index over the creation time for a given order, the system instead suggests that the answer is precomputed using periodically refreshed materialized views.

For the purpose of this evaluation, this transformation is taken a step further by changing the tuple windows to calculate instead the result over hour-long time windows. The reasoning behind this transformation is as follows: TPC-W was initially designed to be run on a single machine and thus the queries were not written expecting arbitrarily high order rates. Since a time window both provides a more semantically consistent result as the system grows and requires strictly more work to maintain than simple tuple windows, this implementation is preferable for a full evaluation.

The first part of the evaluation looks at the 99th percentile response time for each of the web interactions as datasize grows. Figure 5.6 shows that, even after adding the two analytic web interactions, the response time remains virtually constant, independent of the size of the data.

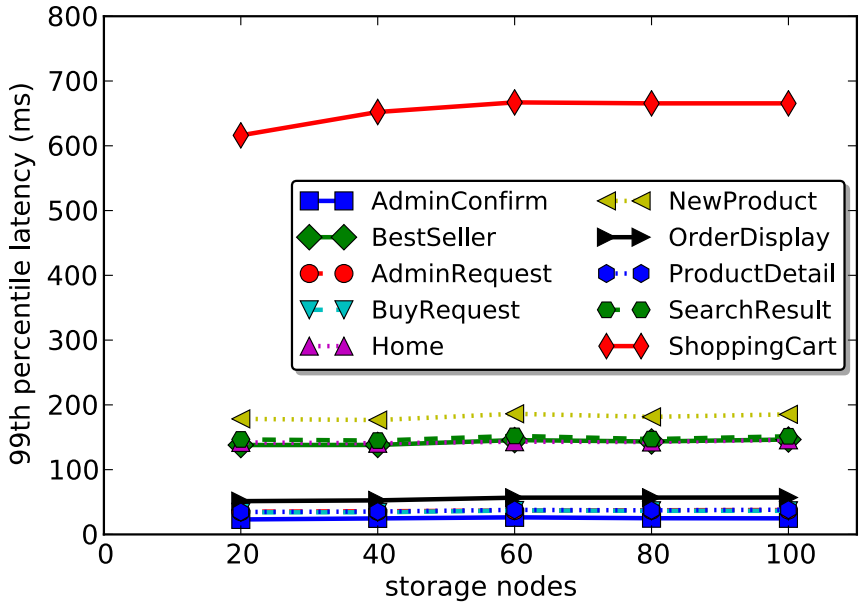

Figure 5.6: The response time for maintaining the IMVs for the TPC-W workload increases slightly initially, but it eventually levels off due to the limitations imposed by the the scale-independent invariants.

Only the ShoppingCart web interaction shows any noticeable change in performance with datasize. This initial increase in latency occurs because at small cluster sizes fewer machines need to be contacted in order to answer the query. However, due to the upper bound on the total number of operations that will be performed by any query, the latency eventually levels off as query execution begins to touch the maximum possible number of machines.

Since the execution of the two analytic queries in the TPC-W benchmark requires maintenance to be performed as data is added into the database, it is also important to measure the latency of the web interaction that performs this maintenance. Figure 5.7 shows that, due to the invariants enforced by PIQL's optimizer, the response time for delta query execution is bounded even as the size of the cluster increases. While the response time exhibits an initial increase in latency similar to the `twoTags` query or the ShoppingCart interaction, the strict upper bound on the number of operations performed in the worst case causes the performance to level off eventually.

5.7.1 Latency of Parallel View Refresh

The performance of the incremental maintenance that was analyzed in the previous section is not the only precomputation that must occur to enable scale-independent execution of these analytic web interactions. Specifically, since the results are hash partitioned across the cluster to avoid hotspots, the actual results for these two web interactions also need

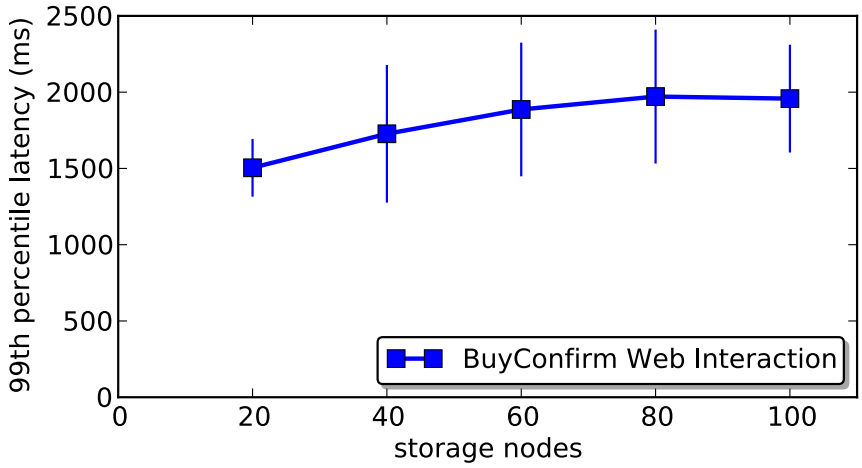

Figure 5.7: The response time for maintaining the IMVs for the TPC-W workload increases initially, but eventually levels off due to the limitations imposed by the the scale-independent invariants.

to be periodically computed in parallel. Figure 5.8 shows that since each partition can be processed in parallel, the overall time taken for the refresh step increases only slightly (less than a second) as the system scales from 20 to 100 machines.

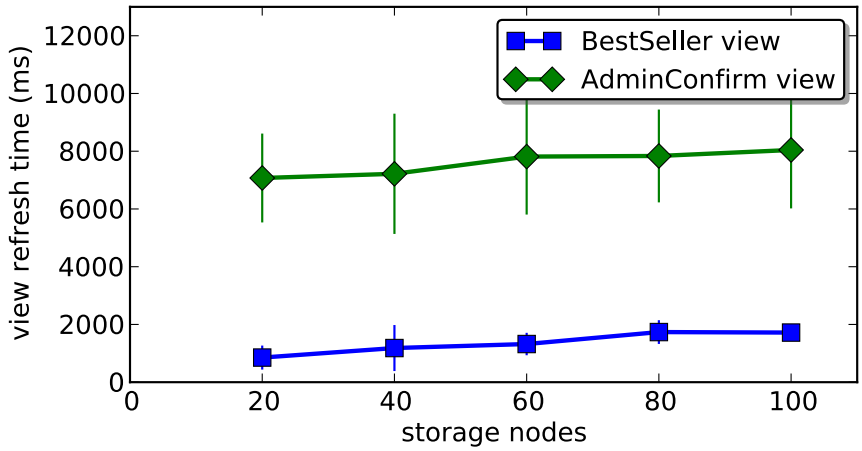

Figure 5.8: Since the amount of serial work per update stays constant independent of the size of the cluster, the latency for periodic view refresh remains nearly constant.

5.8 Summary

Precomputation can often enable the scale independent execution of queries that have no scale-independent physical plans in SI-0 or SI-1. The PIQL optimizer will automatically recognize such queries and construct an appropriate IMV. However, it is important to avoid cases where the storage and maintenance cost of an IMV could cause scaling problems in production. To avoid the creation of potentially unscalable views, PIQL view selection system uses novel static analysis algorithms that ensure that bounds exist on the size and update costs for all created views.

Using these computation-shifting techniques greatly expands the space of scale independent queries. Now that I have described a range of strategies for the scale-independent relational query execution, the next chapter continues with a discussion of the infrastructural support that these techniques leverage in order to provide predictable overall execution time.

Chapter 6

Distributed Storage Manager Support for Scale Independence

6.1 Introduction

PIQL's query execution engine is designed to leverage the scalability of existing distributed storage systems. However, during the construction of the PIQL system, it was found that no freely available storage systems implemented the features required for a full evaluation of scale-independent query processing. This chapter describes the SCADS[1] key/value store [9], which was developed to address these concerns. In addition to describing the functionality of SCADS, I differentiate those aspects of the design that are key to providing scale independence from those which simply help to reduce query processing latency.

This chapter starts with a high-level description of the storage operations that are required by the PIQL execution engine. Next, Section 6.3 describes the overall architecture of SCADS. In particular, this section focuses on the division of responsibility between storage nodes and the SCADS client library. Building on this foundation, Section 6.4 evaluates the importance of asynchrony in communication between these two components. SCADS is only able to provide consistent performance for storage operations when data is evenly distributed across many machines. Section 6.5 describes the API for managing this distribution. Key to the performance of any distributed system is the trade-off that is made between consistency, availability, and partition tolerance [34]. Towards this end, Section 6.6 discusses how PIQL builds on the eventual consistency provided by SCADS to ensure all secondary structures used in query execution are maintained consistently. Finally, Section 6.7 describes the infrastructure used in this thesis for running experiments on large clusters of machines. This section is aimed primarily at readers who wish to recreate or extend the evaluation presented in this thesis and can be safely skipped by others.

[1] Short for Scalable Consistency Adjustable Data Storage.

6.2 Supported Operations

SCADS supports two types of operations: Those that touch a bounded number of servers, and those that run requests on all machines in parallel. Each operation is performed against a *namespace*. A namespace is a logical collection of data that have the same schema. PIQL uses different namespaces for each relation, index, or materialized view that is present in an application.

6.2.1 Bounded-Time Operations

SCADS supports common key/value operations for accessing and modifying data in the system. This section describes the semantics of each of these five basic operations. Each basic operation touches a fixed number of servers and is expected to return in a bounded amount of time, independent of the number of machines in the cluster.

`get(key): value` A `get` takes a full key and returns a single value. The get operation is used by PIQL for the IndexForeignKeyJoin operation, since the whole key for the joined tuple is known. While this operation could be emulated by a `getRange`, it is included for efficiency reasons. This efficiency comes from the simpler operation used to lookup a single value, which does not require opening a cursor in the underlying storage system.

`put(key, value)` A `put` request updates the value stored for the specified key. If the value does not exist in the system, then a new record is created. If no value is specified, then the record with that key is deleted. PIQL translates `INSERT` operations into put requests.

`testAndSet(key, oldValue, newValue): success` Test and set conditionally changes the value stored for the specified key to `newValue`, but only if the currently stored value is equal to `oldValue`. By leaving `oldValue` blank, PIQL can ensure that the update is only performed if the key was not previously stored in the system. PIQL uses the operation to enforce uniqueness constraints when adding new records to the database.

`getRange(startKey, endKey, limit, ascending): key/value pairs` A `getRange` operation returns up to `limit` key/value pairs that fall in-between `startKey` and `endKey`. The key/value pairs are returned in sorted order, either ascending or descending, depending on the value of the `ascending` parameter. Additionally, if only a prefix of the fields present in the key are specified, the `startKey` and `endKey` are padded with the minimum and maximum values respectively. Together these semantics make it possible for PIQL to use `getRange` requests to perform index scans. For example, consider the `twoTag` materialized view from Section 2.3.2, which has the following schema: `TwoTags(tag1, tag2, timestamp, docId)`. PIQL can retrieve the ten

most-recent documents with the tags "lady" and "gaga" by issuing the following request: `getRange({lady, gaga, ,}, {lady, gaga, ,}, 10, false)`.

`incrementField(key, field, delta)` An `incrementField` request updates the record with the specified `key` by adding `delta` to the specified `field`. This operation could also be performed by a `get` followed by a `testAndSet`; however, this method adds the additional latency of a server round trip. Additionally, multiple increment fields can be coalesced as a result of the commutativity of this operation.

6.2.2 Scale-Dependent Operations

Not all operations supported by SCADS touch a fixed number of servers. For example, the operations that are used to periodically refresh views maintained under SI-3 require scanning large amounts of data in parallel on many different storage nodes. In order to support this type of update efficiently, SCADS provides the following two operations:

`topK(startKey, endKey, ordering, k): result` The `topK` operation is performed by sending requests to one replica in each partition between startKey and endKey. Each partition is of roughly a fixed size and the operation proceeds in parallel on each machine. Therefore, in general, topK operations take roughly a fixed amount of time to complete independent of the amount of data in the system. However, the number of tuples that must be sent back to the client might grow with the size of the cluster. Fortunately, as shown in the evaluation from Section 5.7, this growth resulted in a negligible slowdown as the side of the cluster grew.

`groupedTopK(startKey, endKey, grouping, ordering, k, target)` The grouped-topK is similar to the previous operation but instead calculates the top K records for each each unique value of the specified `groupingFields`. For efficiency, the grouping fields must be a prefix of the key. This restriction ensures that the tuples for any given group are contiguous, thus avoiding the need for inter-partition communication.

The number of groups is proportional to the cardinality of the grouping attributes, thus instead of each partition returning the topK tuples to a single client machine, the results are instead inserted into a target namespace. As long as this target namespace is evenly partitioned across the cluster, the inserts will be spread out to many different machines.

To better understand the operation of the `groupedTopK`, consider the following view used to count book orders per subject in the TPC-W benchmark.

subject	*itemID*	*count*
computers	111	1
computers	112	999
computers	113	2
computers	114	0
poetry	115	10
poetry	116	1000

The operation `groupedTopK({,,}, {,,}, [subject], [count], 1)` would return the tuples for items 112 and 116.

Another important set of advanced operations involve changing the placement of data in the cluster. This placement is critical to scale independence, as machines that are over capacity are unable to handle requests in a timely manner and can hurt the overall performance of query execution. Section 6.5 describes these operations in more detail.

6.3 Architecture Overview

Figure 6.1 shows the high-level architecture of the SCADS/PIQL system. At the top of the stack lie the components of the PIQL relational engine. Beneath PIQL lies the client portion of the SCADS key/value store (Section 6.3.2). The client communicates with the storage nodes (Section 6.3.1) using a message passing abstraction (Section 6.4.2).

All of the components run on top of Mesos [46], a cluster scheduling system, which allows more nodes to be dynamically added to the SCADS cluster as the workload demands. Coordination between SCADS components, such as storing the routing table and distributing metadata, is performed using ZooKeeper [48]. Based on Google's Chubby [18], ZooKeeper acts partly as a lock manager and partly as a consistent distributed filesystem.

6.3.1 Storage Nodes

At the base of the SCADS/PIQL database system architecture lie the storage nodes. Berkeley DB (BDB) is used as the backing store and is responsible for persistence of data. The storage nodes are designed to be relatively simple and communicate primarily with the clients instead of amongst themselves. The only exception to this rule is the bulk data transfer operations. When changing the layout of data for a given cluster, servers use an efficient bulk transfer protocol to directly move data between nodes.

6.3.2 Client Library

A majority of SCADS functionality, including concerns such as consistency and index management, is implemented as part of the client library. In Figure 6.1, the client library consists of the red layers, "Trigger Support" through "Range Routing".

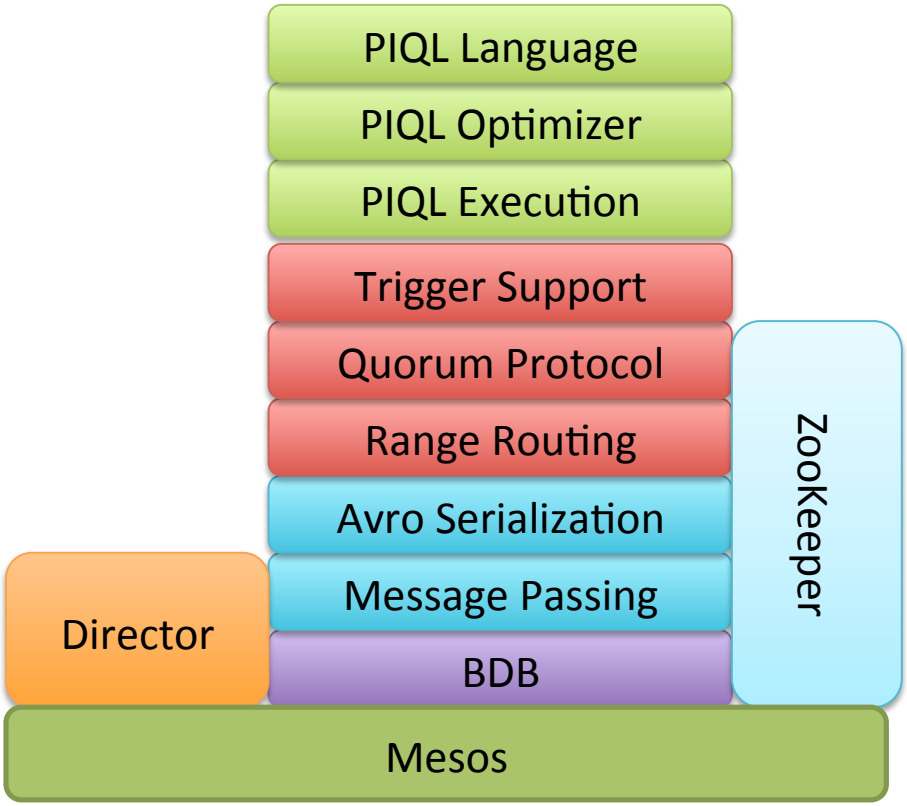

Figure 6.1: The architecture of the SCADS key/value store.

Different modules are implemented as Scala *traits*, allowing the developer to mix and match the desired storage functionality for a given application. This development style is known as the cake pattern [87]. For example, depending on the type of queries that are required, the developer could choose either range partitioning or hash partitioning of data. For the purpose of the experiments presented in this thesis, the following configuration was

utilized:

```
class PairNamespace[Pair <: AvroPair : Manifest](
    val name: String,
    val cluster: ScadsCluster,
    val root: ZooKeeperProxy#ZooKeeperNode)
  extends Namespace
  with SimpleRecordMetadata
  with ZooKeeperGlobalMetadata
  with DefaultKeyRangeRoutable
  with QuorumRangeProtocol
  with AvroPairSerializer[Pair]
  with RecordStore[Pair]
  with CacheManager[Pair]
  with index.IndexManager[Pair]
  with index.ViewManager[Pair]
  with DebuggingClient
  with PerformanceLogger[Pair]
  with NamespaceIterator[Pair]
  ...
```

The handling of metadata in the cluster as well as the partitioning of data across nodes are specified by the top three traits in the namespace declaration (`SimpleRecordMetadata`, `ZooKeeperGlobalMetadata`, `DefaultKeyRangeRoutable`). The `QuorumRangeProtocol` trait indicates that consistency will be managed using a quorum strategy [81]. Next, the traits `AvroPairSerializer` and `RecordStore` specify how records will be serialized for storage and transmission over the network. Other functionality required by PIQL, such as index and view maintenance, are then added to the system through the addition of the `IndexManager` and `ViewManager` traits, respectively. Finally, debugging support for both performance and correctness is provided by the final three traits.

6.4 Communications Layer

Between the storage nodes and client nodes lies the SCADS communication layer. This section discusses the details of serialization and request handling, focusing on how SCADS design choices both enable the functionality required by PIQL and decrease overall query response time.

6.4.1 Serialization

Many key/value stores do not specify a particular data serialization format. Instead, these systems model data as simple binary strings. This approach is sufficient for simple operations

like `get` and `put`, and allows for increased flexibility by giving the higher layers the option to choose the most appropriate representation of data.

Unfortunately, there are many important operations that are not possible when the storage system is unable to understand the semantics of the data being stored. For example, operations such as `incrementField` cannot be performed if the storage system does not have access to the individual attributes for a given tuple. Additionally, operations such as `groupedTopK`, which are necessary for SI-3, also require understanding the schema of the data being stored.

For this reason, SCADS was designed to use Apache Avro [76] serialization to store and transmit data. Similar to other cross-language serialization schemes, such as Google Protocol Buffers [64] and Thrift [78], Avro allows developers to define schema using a language-independent interface definition language (IDL). This IDL allows the specification of record schemas, each of which can contain strongly typed attributes, arrays, and even other nested records. Using this description, the Avro libraries are capable of quickly and compactly serializing a tuple into a string of bytes.

SCADS takes this concept one step further by providing language integrated schema definition. Using a plug-in to the Scala compiler, SCADS scans the code of an application for classes that are annotated with a special marker, *AvroPair*, and adds methods allowing for efficient binary serialization. For example, the schema for the twoTags query (Section 3.3.1) can be represented as follows:

```
case class Document(var docId: Int, var timestamp: Long) extends AvroPair {
  var text: String
}
case class Tag(var docId: Int, var tag: String) extends AvroPair
```

This feature allows developers to avoid the mismatch between SQL and the programming languages type system (often referred to as an *impedance mismatch*) that is typically associated with relational database interface libraries, such as JDBC [68].

6.4.2 Message Passing

While orthogonal to scale independence, allowing asynchrony for remote requests is key to meeting the strict SLOs common for web applications. Specifically, it is very important that the execution engine is capable of issuing many requests to the key/value store in parallel. Similar to the concept of *asynchronous iteration* [35], parallel requests can reduce overall response time by by overlapping the wait time. This functionality is critical to performance, because sending the requests individually wastes valuable execution time while waiting for requests to return. Figure 6.2 illustrates the effect on query response time when requests are made in parallel instead of serially.

To enable this performance optimization, the SCADS communication system supports the following three different types of remote requests.

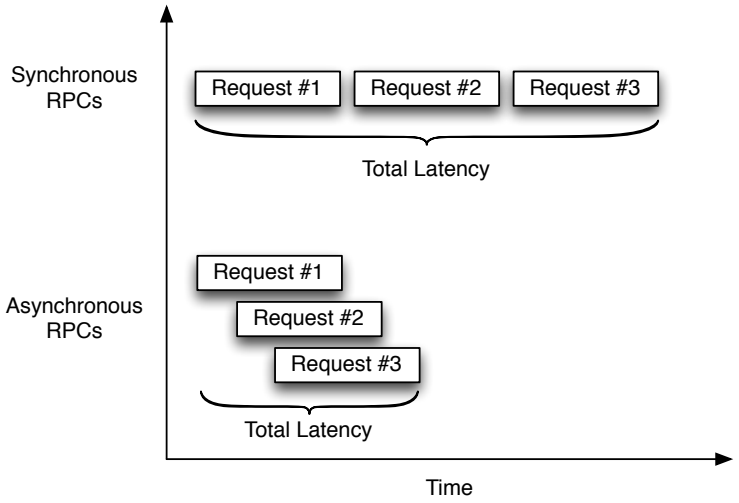

Figure 6.2: An illustration of overall query response time with synchronous and asynchronous RPCs.

Synchronous When only a single request needs to be made or when simplicity is more important than performance, the communication system allows the developer to make standard blocking RPC calls. Each remote request must result in exactly one response. A user configurable timeout is used to avoid stalling forever in the case of lost requests.

Asynchronous Futures Since blocking while waiting for a response can waste significant amounts of execution time, the SCADS communication layer also provides the option to receive a *future* as the result of a remote call. Using this feature, many messages can be sent simultaneously, returning a collection of futures to the developer. Later, when the result of the request is required, the future can be inspected to determine if a response has been received. Library functions are also provided to operate on collections of futures. For example, when implementing a quorum protocol, it is useful to be able to send several requests to different machines and block until at least a majority of the machines have responded.

Custom Message Handlers Finally, the communication system allows the developer to write lightweight custom message handlers that process the responses to an arbitrary number of remote requests in an event-driven manner. These handlers can either process messages in parallel on multiple threads or provide *actor* semantics by ensuring the message handler code is never run for more than a single message at a time. Custom message handlers ease the implementation of complex multiphase protocols, such as MDCC [53].

6.4.2.1 Evaluation of Asynchrony

The flexibility of the communication system allows for the implementation of different execution strategies, the choice of which can have a significant effect on the overall response time for an application. This effect can be evaluated by running the TPC-W benchmark on a fixed cluster size using each of the different execution strategies. In this experiment, three different variants of the PIQL execution engine are run on a cluster with 10 storage nodes and 5 client machines. The amount of data and the length of the experiment are kept the same as in the TPC-W scale experiment.

Figure 6.3 shows the 99th-percentile latency for each execution strategy. The first strategy, called the Lazy Executor, operates in a similar fashion to a traditional relational database system by requesting a single tuple at a time from the key/value store, blocking until a response is received. The second strategy, called the Simple Executor, utilizes the extra limit hint information provided by the optimizer to request data in batches from the key/value store; however, it waits for each request to return before issuing the next. The final strategy, called the Parallel Executor, uses the extra limit hint information and issues all key/value store requests in parallel for a given remote operator. The results show the importance of both the limit hint information and the intra-query parallelism enabled by the SCADS communication layer.

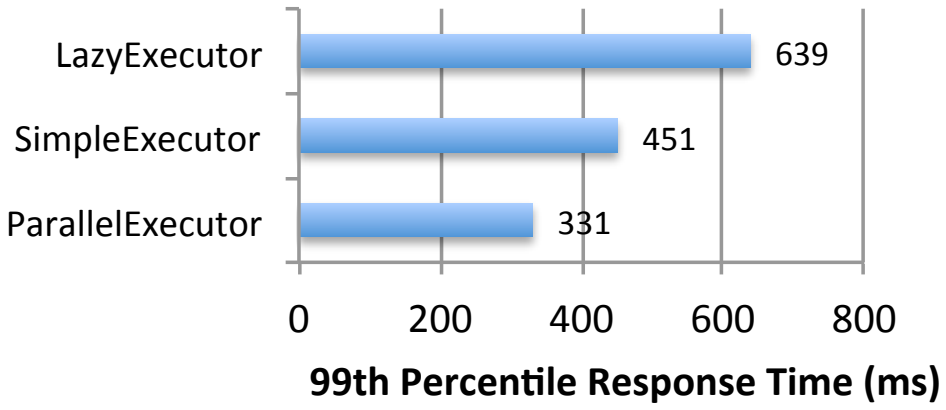

Figure 6.3: TPC-W 99th Percentile Response Time By Varying the Execution Strategy.

6.5 Data partitioning

The ability to spread data out over many machines is critical to building a scale-independent system. PIQL is only able to maintain consistent response time as the data and system size

grow because SCADS is capable of evenly spreading data and requests across a cluster of increasing size.

At the core of SCADS data layout abstraction lies the idea of a *partition*. Each partition is responsible for the data in a namespace that falls between the specified *start key* and *end key*. Multiple replicas can exist for each partition, allowing for greater availability and performance. As suggested by Jeff Dean [30], an architect of many of Google's distributed system, each machine is responsible for $\sim 10 - 100$ partitions. This level of partitioning balances the need for fine-grained control of placement and workload information with the overhead of managing individual partitions.

6.5.1 API for Changing Data Partitioning

In order to allow for experimentation with different data layout policies, SCADS provides a programmatic API for modifying the placement of data. This API allows developers to change the distribution of data across many machines while automatically handling concerns such as transferring data, updating the routing table, and notifying clients of the update. SCADS provides developers with the following four data placement primitives.

`replicate(partitionId, newStorageNodeId): newPartitionId` A replicate request takes an existing partition and a storage node as arguments. Data from the existing partition is copied to the new server, and the routing table is updated to include the new partition.

`delete(partitionId)` A delete request takes the address of an existing replica. The routing table is updated to remove the specified replica and the data is lazily deleted. Data can be moved from one server to another by issuing a replicate request followed by a delete request.

`split(splitPoint): newPartitionIds` When a partition grows too large, it can be split. A split request takes as input a value for the new *split point*. All replicas that contain this point are then split into two new partitions and the address of these new partitions is returned to the developer. The routing table is also updated to reflect the presence of these new partitions.

`merge(splitPoint): newPartitionIds` Conversely, the developer can also merge partitions by specifying a split point that should be removed from the routing table. The replicas on either side of these points are merged, and the addresses of the newly merged partitions are returned to the developer.

6.5.2 Autonomic Control

When performing the evaluation presented in this thesis, the above partitioning API was used to statically distribute data evenly throughout the cluster, ensuring consistent response time

for low-level storage operations. However, Trushkowsky, et al. [84] demonstrated that similar performance can be obtained using a more autonomic approach. The SCADS director uses the data partitioning API to dynamically change the distribution of data based on statistics collected on the current workload.

Their technique, known as *model predictive control*, is based on building performance models of the expected performance given a workload. This strategy avoids unnecessary churn that results from attempting to make placement decisions using a noisy indicator like 99th percentile response time. Given a previously trained performance model and statistics about the current workload, the director makes decisions about how to distribute data in order to maintain the specified SLO.

6.6 Consistency

The SCADS storage system currently supports only eventual consistency [75]. While efficiently providing full ACID transactions in a distributed system is out of the scope of this thesis, it is important that PIQL does not allow permanent inconsistency in the secondary structures used for query execution. Therefore, PIQL must build on the primitives provided by SCADS to give developers a reasonable environment for building applications. For example, atomic operations such as test-and-set can be used to deliver the expected uniqueness semantics required to implement benchmarks such as TPC-W.

Other consistency requirements, however, can be more complicated. For example, maintaining secondary indexes requires a form of atomicity, as a crash might cause indexes never to be updated, which would not even provide eventual consistency. PIQL avoids this potential inconsistency using the following index maintenance protocol. First, all new keys are inserted to the namespace that holds the secondary index. Next, the actual value of the record is updated. Finally, all stale secondary index entries are deleted from the secondary index. This protocol ensures that, in the worse case, a crash could result in the presence of dangling pointers in the secondary index. Fortunately, these can be detected and garbage collected lazily using a modified form of read repair [47].

Cardinality constraints present an additional challenge in the face of eventual consistency. PIQL ensures the cardinality constraints on relationships using the following protocol: After inserting an item, the system checks the cardinality constraint using a count range request. If the total count returned is less than the constraint, the insert is considered successful. If not, the inserted record is deleted. Note that this protocol might temporarily violate cardinality constraints for concurrent insertions.

Correctly maintaining views in the face of concurrent updates is the final challenge resulting from SCADSs eventual consistency semantics. PIQL's static analysis of materialized views is generally orthogonal to the mechanism used to perform maintenance. Thus, these techniques could be used together with different consistency maintenance strategies, such as online maintenance [58, 62] or deferred view maintenance [66, 28], each of which provide their own consistency semantics. Additionally, for applications with strong consistency re-

quirements, a locking-based view maintenance mechanism can be used [62, 58]. However, depending on the data access patterns involved, such locking-based techniques might create contention points and, therefore, potentially violate the goal of predictable performance at scale.

As an alternative to locking-based techniques, many deferred view maintenance techniques, such as 2VNL [66], perform optimistic view maintenance. This approach frequently yields better performance by avoiding contention points while providing a weaker consistency guarantee (e.g., snapshot isolation). Finally, eventually consistent view maintenance techniques [62] provide the lowest overhead per update, but often at the price of a few concurrent updates never being reflected in the view (see also view maintenance anomalies [55]).

The current implementation of PIQL uses the simplest technique, relaxed view maintenance. This strategy ensures that all updates are eventually reflected in the view using the following procedure: Delta queries execute under relaxed consistency, while background batch jobs check for and repair any inconsistencies that may arise by periodically reconstructing the entire materialized view. The frequency of the execution of these batch jobs can be tuned to meet the needs of a specific application. This approach represents a balance between the needs for low-latency update propagation, high availability, and consistency. Clearly, the efficiency of PIQL could be improved by only checking for inconsistencies in those sections of the view that have changed recently as done in [66]. This optimization, as well as other mechanisms for eventually-consistent, distributed, incremental view maintenance, represent interesting future research problems.

6.7 Deploying on Large Clusters

Another key feature of SCADS, which enabled much of the evaluation presented in this thesis, is the built-in support for deploying the system on large clusters of machines. This section briefly describes *deploylib*, the library used to run SCADS on large clusters of EC2 machines. Readers who would like to experiment with the code presented can download SCADS, PIQL, and all the code used for evaluation from https://github.com/radlab/scads. This section is not intended as a full overview of the features of deploylib, but merely a starting point for those wishing to rerun the experiments presented in this thesis. A more thorough description of deploylib can be found at: https://github.com/radlab/SCADS/wiki/Deploylib.

The evaluation of SCADS and PIQL primarily relies on two components of deploylib. The first component, `deploylib.mesos.Cluster`, uses ssh to set up and manage a cluster of machines running the Mesos cluster scheduling system. The second, the service scheduler, integrates with the sbt [80] build system to gather up all the required executable files and launch a given program on Mesos. As an example, consider the following code segment, which starts up a Mesos cluster on a set of EC2 machines and then runs a SCADS storage node.

```
~/Workspace/radlab/scads> sbt mviews/deploy-console
```

```
[info] Loading project definition from ~/Workspace/radlab/scads/project
[info] Set project to scads (in build file:~/Workspace/radlab/scads/)
[info] Starting scala interpreter...

import deploylib._
import deploylib.ec2._
import deploylib.mesos._
allJars: Seq[java.io.File] = List(deploylib_2.9.1-2.1.4-SNAPSHOT.jar, ...)
Type in expressions to have them evaluated.
Type :help for more information.

scala> cluster.setup(1)
INF [20130204-15:04:00.892] ec2: Updated EC2 instances state
INF [20130204-15:04:01.257] mesos: Starting a master.
INF [20130204-15:04:02.876] deploylib: Waiting for instance i-fe2b35a7
INF [20130204-15:04:03.631] deploylib: Waiting for instance i-f02b35a9
INF [20130204-15:04:04.168] deploylib: Waiting for instance i-f22b35ab
INF [20130204-15:05:35.188] ec2: Uploadingdeploylib_2.9.1-2...
...

scala> import org.apache.zookeeper.CreateMode
scala> val scadsRoot = cluster.zooKeeperRoot.createChild("testCluster")
scala> val engTask = ScalaEngineTask(scadsRoot.canonicalAddress).toJvmTask
scala> cluster.serviceScheduler !? RunExperimentRequest(engTask :: Nil)
```

The previous code segment starts by bringing up the deployment console. A deployment console provides support for interactively running commands on a cluster of machines. Additionally, it integrates with the build system for the current project to ensure that the latest version of any application code, including dependencies, is shipped to the servers of the cluster for execution. Once the console is launched, the example next launches a cluster of machines on Amazon's EC2 using the `cluster.setup` command. A cluster consists of a ZooKeeper quorum, a mesos master, and the specified number of machines. After the cluster is started, the final three lines of code create a coordination point for the scads cluster in ZooKeeper and then start a scads storage node instance by submitting a `RunExperimentRequest` to the serviceScheduler. While this is only a simple example, all of the experiments presented in this thesis can be run using similar techniques.

6.8 Summary

The SCADS storage system combines several techniques that enable the execution of complicated applications that continue to meet their SLOs as the data grows rapidly. Some of

these techniques are critical to scale independence. For example, the bounded time from operations (Section 6.2.1) are used by the PIQL execution engine to answer queries under execution level SI-0 – SI-2. Additionally, when queries are answered under SI-3 using periodic view refresh, parallel operations such as `topK` and `groupedTopK` are required. Finally, all of the performance guarantees provided by SCADS for these operations are only valid when data is evenly distributed across the cluster using the data partitioning API.

In addition to providing the primitives required for scale-independent query execution, SCADS also leverages several techniques that greatly reduce the absolute time taken to execute queries. Specifically, the ability to make asynchronous requests to many storage servers in parallel significantly diminishes the amount of time wasted waiting for requests to return from remote machines. Taken together, the scale-independent optimization provided by PIQL and the scalability of the SCADS architecture make it possible for developers to easily build applications that perform predictably even as data sizes grow rapidly.

Chapter 7

Conclusion

This chapter begins with a review of the primary contributions of this thesis. Next, I discuss some of the primary limitations of PIQL's implementation of scale independence, as well as some possible future directions for scale independence research.

7.1 Contributions

I began this thesis by formally defining the characteristics of scalable queries. I used this formalism to construct different classes of SQL queries, segmenting them based on the techniques that are required to ensure they scale gracefully. Building on this theoretical foundation, I presented PIQL, a first attempt at building a scale-independent RDBMS. PIQL is capable of determining the inherent scalability of all the queries in an application. Furthermore, if these queries have been determined scalable, the underlying distributed architecture can ensure that they execute with predictable performance as both data and the number of machines grow. This accomplishment required rethinking the implementation at all layers, including the query language, optimizer, view selection system, as well as the underlying storage system.

The PIQL query compiler uses static analysis to allow only query plans where it can calculate a bound on the number of key/value operations to be performed at every step in their execution. Therefore, in contrast to traditional query optimizers, the objective function of the query compiler is not to find the plan that is fastest on average. Rather, the goal is to avoid performance degradation as the database grows. To avoid choosing plans that perform too many storage operations, PIQL employs a worst case performance prediction model. If the PIQL compiler cannot create a bounded plan for a query, it warns the developer and suggests possible ways to bound the computation.

PIQL will warn the user if any queries cannot be deemed scale independent, in order to prevent performance problems at runtime. Therefore, to support real-world applications, I had to enhance the systems capabilities to handle SQL queries where it is simply impossible to bound the number of storage operations required if all processing is performed on-demand.

I found that it is often possible to answer such queries safely at scale by leveraging incremental precomputation, effectively shifting some query processing work from execution time to insertion time. I formally defined the classes of SQL queries where precomputation fundamentally changes the worst case execution cost at scale. Understanding the characteristics of these classes allowed me to construct a scale-independent view selection and maintenance system within PIQL.

7.2 Limitations

While the techniques presented in this thesis are more than sufficient to implement several complex applications, it is my belief that this work represents only the tip of the scale-independence iceberg. Specifically, the scale independence execution levels supported by PIQL most likely represent only a subset of all possible mechanisms for executing queries with predictable performance, even as the data grows by orders of magnitude. Since not all possible execution strategies are supported in the current implementation, there exist queries which the system will wrongly deem unscalable.

Traditional database systems take a different approach by allowing users to run any query, even when an efficient execution strategy cannot be found. PIQL, in contrast, can make promises about the worst case execution time, but these assurances come at the cost of possible false negatives. False negatives occur when a query that could theoretically be executed scale-independently is rejected by the system. They can occur for two different reasons. The first reason is that the optimizer might be missing necessary execution techniques, such as range trees [22], as discussed below. In other words, the query belongs to a scale independence execution level that has not yet been defined. The second reason is the built-in assumption that all tables are going to grow significantly in size. While this assumption is clearly safe, it can result in the system being overly cautious about what queries can be executed, since in practice many tables are not going to grow arbitrarily large.

While rejecting false negatives limits the ability of PIQL to handle all queries as a traditional RDBMS would, I feel that it is a much better approach than accepting any false positives, which would erode developer confidence in the performance guarantees and drive them back to NoSQL inefficiency.

7.3 Future Challenges

Going forward, research into *scale-independent thinking* should strive to increase the space of queries which can be analyzed, optimized, and executed safely at scale. I see several specific areas in which progress against this metric can be made.

7.3.1 Other Index Structures and Execution Strategies

Increasing the number of index structures and execution strategies that are available to the optimizer could increase the set of queries that can be run scale-independently. For example, R-Trees could enable other types of queries to be run while only performing a bounded number of operations. PIQL is currently unable to execute queries that involve inequalities over multiple attributes as doing so might require scanning over an arbitrary number of tuples from a standard secondary index. Fortunately, these other structures could make such queries safe to run by extending SI-1.

7.3.2 Eventually Consistent View Maintenance

Precomputation is essential to alleviating many possible scaling problems. Unfortunately, current view maintenance techniques are woefully inadequate, as they make many assumptions about consistency in order to ensure correct maintenance.

Since these assumptions are often unreasonable in a distributed system, the current implementation of PIQL is only able to perform eventually consistent view maintenance by periodically recomputing the contents of each view from scratch. While this strategy avoids permanently lost writes, it comes at the cost of significant computational overhead.

Instead, an ideal system would be capable of updating views incrementally, detecting conflicts and repairing them using a more efficient process. Supporting scalability through eventually consistent view maintenance is only going to become more important as geographic distribution of data becomes more pervasive. Therefore, developing a system with such capabilities will be important to continue to support SI-2 and SI-3 for global data management situations.

7.3.3 Big Data and Estimation

The scale-independent execution techniques described in this thesis focus on executing queries where it is possible to return the exact results while bounding the amount of processing work required. With the increasing popularity of "Big Data" analytics, it is often desirable to be able to bound the response time for more complex analytic queries even when it is not possible to bound the total amount of work required to compute an exact answer.

Fortunately, there has been a significant amount of work on estimating the answer to relational queries using techniques such as sampling [1]. Switching to such a strategy, however, raises several questions including: "In what cases can sampling be used to bound strictly the amount of work required to answer a query?" and "How does error change for different queries as the amount of data grows by orders of magnitude?" Expanding the concept of scale-independence to answer these questions is likely a PhD thesis in itself.

7.4 Final Remarks

The advent of the Internet has ushered in a new paradigm of software development, one in which interactive applications can be developed in a single weekend and deployed without capital investment to a user base that will ideally grow exponentially. The success of these new applications is often evaluated by their ability to handle such intense scaling gracefully over years of worldwide growth, since exploiting network effects increases the value of the experience that they offer to their users. A "success disaster" befalls applications whose designers fail to predict the effects that ceaseless growth will render on their performance and user experience, often leading to the untimely demise of the application at the hands of a less-bottlenecked competitor.

Unfortunately, the software development tools available to system architects embarking on such projects lack the ability to provide insight into inefficiencies that will threaten performance as the application grows in popularity. Data management is the heart of the scalability challenge, since most other components of web applications are stateless and, hence, trivially scalable. Examining the current landscape of data management solutions reveals that none of them support both ongoing developer productivity and long-term scalability robustness.

Traditional RDBMS tools provide developers with a consistent way of interacting with data, hide challenges associated with concurrency and fault tolerance, and support declarative query languages that make feature development faster and more agile. The data independence provided by relational database systems has proven to be a huge boon to developer productivity. By separating the semantics of data access from the details of performing it efficiently, the relational model allows developers to focus on adding features while often providing better performance than hand-coded implementations. Even more importantly in the modern world of agile development, this separation makes it easy to change the representation of the data without impairing the functionality of the application.

However, RDBMS do little to prevent "success disasters". In fact, many developers feel that the declarative, high-level programming interface provided by SQL database systems invites disaster by enabling them to inadvertently write queries prone to scalability problems. As the actual mechanism for execution is divorced from specification given by the developer, it can be very difficult for developers to reason about performance problems. Furthermore, as database statistics change with data growth, it is not uncommon for new query execution strategies to be chosen. This inherent unpredictability can make it virtually impossible to determine how application performance will change as popularity increases.

In contrast, NoSQL data management solutions provide developers with key/value stores that are capable of scaling performance with machine count nearly as easily as stateless system components. Developers seem willing to accept the straightforward pain of hand-coding imperative queries against such distributed key/value stores, even at the cost of losing physical and logical data independence. While changes to the data model in this paradigm often require time-consuming rewrites of application-level code, the NoSQL developer can feel confident in the performance of the current implementation.

Unfortunately, these developers must still choose an arbitrary scalability target and en-

gineer their application to handle that particular level of traffic. This brute-force solution is sub-optimal for two reasons. First, this approach diverts critical development energy and resources planning for traffic that may never arrive. Second, even if popularity is achieved and the engineering effort is eventually warranted, the chosen scalability target may still be insufficient. So, while popular in the wild, NoSQL solutions add a host of difficulties to application maintainability, all in the name of more predictable performance.

My thesis is that the schism between productivity and predictability is not fundamental to distributed data management. By defining and reasoning about scale independence, a new type of data independence, I have provided tools that ensure that applications will perform predictably as they grow in popularity, while simultaneously preserving the many productivity benefits of the relational model. A scale-independent system is inherently *success-tolerant*, assuring developers that their initial high-level, declarative implementation will survive the massive onslaught of data intrinsic to success on the web.

Bibliography

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 29–42. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465355. URL: http://doi.acm.org/10.1145/2465351.2465355.

[2] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. "Asynchronous view maintenance for VLSD databases". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 179–192. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559866. URL: http://doi.acm.org/10.1145/1559845.1559866.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *VLDB*. 2000.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 496–505. ISBN: 1-55860-715-3. URL: http://dl.acm.org/citation.cfm?id=645926.671701.

[5] Yanif Ahmad and Christoph Koch. "DBToaster: a SQL compiler for high-performance delta processing in main-memory databases". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1566–1569. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687553.1687592.

[6] *Apache CouchDBt*. URL: http://couchdb.apache.org/.

[7] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. "PIQL: success-tolerant query processing in the cloud". In: *Proc. VLDB Endow.* 5.3 (Nov. 2011), pp. 181–192. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=2078331.2078334.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing*. Tech. rep. UCB/EECS-2009-28. EECS Department, University of California, Berkeley, 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[9] Michael Armbrust, Armando Fox, David A. Patterson, Nick Lanham, Beth Trushkowsky, Jesse Trutna, and Haruki Oh. "SCADS: Scale-Independent Storage for Social Computing Applications". In: *CIDR*. www.cidrdb.org, 2009.

[10] Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. "Generalized scale independence through incremental precomputation". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 625–636. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465333. URL: http://doi.acm.org/10.1145/2463676.2465333.

[11] Charles Arthur. "Average Twitter user has 126 followers, and only 20go via website". In: *Guardian Technology Blog* (2009). URL: http://www.guardian.co.uk/technology/blog/2009/jun/29/twitter-users-average-api-traffic.

[12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. "Probabilistically Bounded Staleness for Practical Partial Quorums". In: *PVLDB* 5.8 (2012), pp. 776–787.

[13] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In: *CIDR*. www.cidrdb.org, 2011, pp. 223–234.

[14] Amiee Lee Ball. "Are 5001 friends on Facebook one too many?" In: *New York Times* (2010).

[15] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. "Materialized Views Selection in a Multidimensional Database". In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 156–165. ISBN: 1-55860-470-7. URL: http://dl.acm.org/citation.cfm?id=645923.671019.

[16] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently updating materialized views". In: *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*. SIGMOD '86. Washington, D.C., United States: ACM, 1986, pp. 61–71. ISBN: 0-89791-191-1. DOI: 10.1145/16894.16861. URL: http://doi.acm.org/10.1145/16894.16861.

[17] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. "Building a database on S3". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 251–264. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376645. URL: http://doi.acm.org/10.1145/1376616.1376645.

[18] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 335–350. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298487.

[19] Michael J. Carey and Donald Kossmann. "On saying "Enough already!" in SQL". In: *SIGMOD Rec.* 26.2 (1997), pp. 219–230. ISSN: 0163-5808. DOI: http://doi.acm.org/10.1145/253262.253302.

[20] *Cassandra Query Language (CQL) v2.0*. URL: http://cassandra.apache.org/doc/cql/CQL.html.

[21] Stefano Ceri and Jennifer Widom. "Deriving Production Rules for Incremental View Maintenance". In: *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 577–589. ISBN: 1-55860-150-3. URL: http://dl.acm.org/citation.cfm?id=645917.672169.

[22] Moez Chaabouni et al. "The point-range tree: a data structure for indexing intervals". In: *Proc. of ACM CSC*. Indianapolis, Indiana, USA, 1993. ISBN: 0-89791-558-5.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[24] *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 2011.

[25] *CODASYL Data Base Task Group report*. Tech. rep. 1971.

[26] E. F. Codd. "A relational model of data for large shared data banks". In: *Commun. ACM* 26.1 (Jan. 1983), pp. 64–69. ISSN: 0001-0782. DOI: 10.1145/357980.358007. URL: http://doi.acm.org/10.1145/357980.358007.

[27] Jeff Cogswell. *SQL vs. NoSQL: Which Is Better?* URL: http://slashdot.org/topic/bi/sql-vs-nosql-which-is-better/.

[28] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. "Algorithms for deferred view maintenance". In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. SIGMOD '96. Montreal, Quebec, Canada: ACM, 1996, pp. 469–480. ISBN: 0-89791-794-4. DOI: 10.1145/233269.233364. URL: http://doi.acm.org/10.1145/233269.233364.

[29] Evandro Cunha, Gabriel Magno, Giovanni Comarela, Virgilio Almeida, Marcos André Gonçalves, and Fabricio Benevenuto. "Analyzing the Dynamic Evolution of Hashtags on Twitter: a Language-Based Approach". In: *Proceedings of the Workshop on Language in Social Media (LSM 2011)*. Portland, Oregon: Association for Computational Linguistics, 2011, pp. 58–65. URL: http://www.aclweb.org/anthology/W11-0708.

[30] Jeffrey Dean. "Evolution and future directions of large-scale storage and computation systems at Google". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1–1. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807130. URL: http://doi.acm.org/10.1145/1807128.1807130.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[32] *django*. URL: https://www.djangoproject.com/.

[33] Armando Fox and David A. Patterson. *Engineering Long-Lasting Software: An Agile Approach Using SaaS and Cloud Computing, Alpha Edition*. Strawberry Canyon LLC, 2012.

[34] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services". In: *In ACM SIGACT News*. 2002, p. 2002.

[35] Roy Goldman and Jennifer Widom. "WSQ/DSQ: a practical approach for combined querying of databases and the Web". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. Dallas, Texas, USA: ACM, 2000, pp. 285–296. ISBN: 1-58113-217-4. DOI: 10.1145/342009.335422. URL: http://doi.acm.org/10.1145/342009.335422.

[36] Jonathan Goldstein and Paul Larson. "Optimizing queries using materialized views: a practical, scalable solution". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 331–342. ISBN: 1-58113-332-4. DOI: 10.1145/375663.375706. URL: http://doi.acm.org/10.1145/375663.375706.

[37] *Google AppEngine: The Google Query Language*. 2011. URL: http://code.google.com/appengine/docs/python/datastore/.

[38] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals". In: *Data Min. Knowl. Discov.* 1.1 (Jan. 1997), pp. 29–53. ISSN: 1384-5810. DOI: 10.1023/A:1009726021843. URL: http://dx.doi.org/10.1023/A:1009726021843.

[39] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. "Counting solutions to the View Maintenance Problem". In: *IN WORKSHOP ON DEDUCTIVE DATABASES, JICSLP*. 1992, pp. 185–194.

[40] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. "Counting solutions to the View Maintenance Problem". In: *Workshop on Deductive Databases, JICSLP*. Ed. by Kotagiri Ramamohanarao, James Harland, and Guozhu Dong. Department of Computer Science, University of Melbourne, 1992, pp. 185–194.

[41] Ashish Gupta and Inderpal Singh Mumick. "Materialized views". In: ed. by Ashish Gupta and Iderpal Singh Mumick. Cambridge, MA, USA: MIT Press, 1999. Chap. Maintenance of materialized views: problems, techniques, and applications, pp. 145–157. ISBN: 0-262-57122-6. URL: http://dl.acm.org/citation.cfm?id=310709.310737.

[42] Himanshu Gupta. "Selection of Views to Materialize in a Data Warehouse". In: *Proceedings of the 6th International Conference on Database Theory*. ICDT '97. London, UK, UK: Springer-Verlag, 1997, pp. 98–112. ISBN: 3-540-62222-5. URL: http://dl.acm.org/citation.cfm?id=645502.656089.

[43] Patrick Hall, Peter Hitchcock, and Stephen Todd. "An algebra of relations for machine computation". In: *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '75. Palo Alto, California: ACM, 1975, pp. 225–232. DOI: 10.1145/512976.512998. URL: http://doi.acm.org/10.1145/512976.512998.

[44] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. "Implementing data cubes efficiently". In: *SIGMOD Rec.* 25.2 (June 1996), pp. 205–216.

[45] A. Hernando et al. "Unravelling the size distribution of social groups with information theory in complex networks". In: *European Physical Journal B* (2010). DOI: 10.1140/epjb/e2010-00216-1.

[46] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos: a platform for fine-grained resource sharing in the data center". In: *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 22–22. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[47] Andrew C. Huang and Armando Fox. "Cheap recovery: a key to self-managing state". In: *Trans. Storage* 1.1 (Feb. 2005), pp. 38–70. ISSN: 1553-3077. DOI: 10.1145/1044956.1044959. URL: http://doi.acm.org/10.1145/1044956.1044959.

[48] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: wait-free coordination for internet-scale systems". In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[49] Jason Kincaid. *Zuckerberg: Online Sharing Is Growing At An Exponential Rate*. URL: http://techcrunch.com/2011/07/06/zuckerberg-online-sharing-is-growing-at-an-exponential-rate-and-users-are-sharing-4-billion-things-a-day/.

[50] Christoph Koch. "Incremental query evaluation in a ring of databases". In: *PODS*. Ed. by Jan Paredaens and Dirk Van Gucht. ACM, 2010, pp. 87–98. ISBN: 978-1-4503-0033-9.

[51] Yannis Kotidis and Nick Roussopoulos. "DynaMat: a dynamic view management system for data warehouses". In: *SIGMOD Rec.* 28.2 (1999).

[52] Tim Kraska. "Building Database Applications in the Cloud". PhD thesis. ETH Zurich, 2010. URL: http://e-collection.library.ethz.ch/eserv/eth:924/eth-924-02.pdf?pid=eth:924&dsID=eth-924-02.pdf.

[53] Tim Kraska, Gene Pang, Alan Fekete, Michael J. Franklin, and Samuel Madden. "MDCC: Multi-Data Center Consistency". In: *Eurosys (to appear)*. 2013.

[54] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. "Optimization of Nonrecursive Queries". In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 128–137. ISBN: 0-934613-18-4. URL: http://dl.acm.org/citation.cfm?id=645913.671481.

[55] Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, and Jennifer Widom. "Performance Issues in Incremental Warehouse Maintenance". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 461–472. ISBN: 1-55860-715-3. URL: http://dl.acm.org/citation.cfm?id=645926.671684.

[56] Weifa Liang, Hui Wang, and Maria E. Orlowska. "Materialized view selection under the maintenance time constraint". In: *Data Knowl. Eng.* 37.2 (May 2001), pp. 203–216. ISSN: 0169-023X. DOI: 10.1016/S0169-023X(01)00007-6. URL: http://dx.doi.org/10.1016/S0169-023X(01)00007-6.

[57] Xiaohui Long and Torsten Suel. "Three-level caching for efficient query processing in large Web search engines". In: *Proceedings of the 14th international conference on World Wide Web*. WWW '05. Chiba, Japan: ACM, 2005, pp. 257–266. ISBN: 1-59593-046-9. DOI: 10.1145/1060745.1060785. URL: http://doi.acm.org/10.1145/1060745.1060785.

[58] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. "Locking protocols for materialized aggregate join views". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 596–607. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315503.

[59] *memcached*. URL: http://memcached.org/.

[60] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. "Materialized view selection and maintenance using multi-query optimization". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. SIGMOD '01. Santa Barbara, California, United States: ACM, 2001, pp. 307–318. ISBN: 1-58113-332-4. DOI: 10.1145/375663.375703. URL: http://doi.acm.org/10.1145/375663.375703.

[61] M. E. J. Newman. "Power laws, Pareto distributions and Zipf's law". In: *Contemporary Physics* 46 (2005), pp. 323–351. arXiv:cond-mat/0412004. URL: http://arxiv.org/abs/cond-mat/0412004.

[62] M. Tamer Ozsu. *Principles of Distributed Database Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780130412126.

[63] *Play!* URL: http://www.playframework.org/.

[64] *Protocol Buffers - Google's data interchange format*. URL: https://code.google.com/p/protobuf/.

[65] *Proven, real-time scalability*. URL: http://www.salesforce.com/platform/cloud-infrastructure/scalability.jsp.

[66] Dallan Quass and Jennifer Widom. "On-line warehouse view maintenance". In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 393–404. ISBN: 0-89791-911-4. DOI: 10.1145/253260.253352. URL: http://doi.acm.org/10.1145/253260.253352.

[67] Christoph Quix, David Kensche Xiang Li and, and Sandra Geisler. *View Management Techniques and Their Application to Data Stream Management*. IGI Global, 2010.

[68] George Reese. *Database Programming with JDBC and Java, Second Edition*. Ed. by Andy Oram. 2nd. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000. ISBN: 1565926161.

[69] Gary Rivlin. "Wallflower at the Web Party". In: *New York Times* (2006).

[70] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. "Materialized view maintenance and integrity constraint checking: trading space for time". In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. SIGMOD '96. Montreal, Quebec, Canada: ACM, 1996, pp. 447–458. ISBN: 0-89791-794-4. DOI: 10.1145/233269.233361. URL: http://doi.acm.org/10.1145/233269.233361.

[71] *Ruby on Rails*. URL: http://rubyonrails.org/.

[72] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. "How to roll a join: asynchronous incremental view maintenance". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. Dallas, Texas, United States: ACM, 2000, pp. 129–140. ISBN: 1-58113-217-4. DOI: 10.1145/342009.335393. URL: http://doi.acm.org/10.1145/342009.335393.

[73] Eric Schurman and Jake Brutlag. *Performance Related Changes and their User Impact*. Presented at Velocity Web Performance and Operations Conference. 2009.

[74] Robert Scoble. *The you-don't-need-more-friends lobby*. URL: http://scobleizer.com/2007/10/14/the-you-dont-need-more-friends-lobby/.

[75] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. PDIS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 140–149. ISBN: 0-8186-6400-2. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

[76] *The Apache Avro Project*. URL: http://avro.apache.org/.

[77] *The Apache Cassandra Project*. URL: http://cassandra.apache.org/.

[78] *The Apache Thrift Project*. URL: http://thrift.apache.org/.

[79] *The PostgreSQL Database Server Open Source Project on Ohloh : Languages Page*. URL: http://www.ohloh.net/p/postgres/analyses/latest/languages_summary.

[80] *The Simple Build Tool*. URL: http://www.scala-sbt.org.

[81] Robert H. Thomas. "A Majority consensus approach to concurrency control for multiple copy databases". In: *ACM Trans. Database Syst.* 4.2 (June 1979), pp. 180–209. ISSN: 0362-5915. DOI: 10.1145/320071.320076. URL: http://doi.acm.org/10.1145/320071.320076.

[82] *TPC BENCHMARK D - Standard Specification*. Tech. rep. Transaction Processing Performance Council (TPC), 1998. URL: http://www.tpc.org/tpcd/spec/tpcd_current.pdf.

[83] *TPC Benchmark W (Web Commerce)*. Tech. rep. Transaction Processing Performance Council (TPC), 2002. URL: http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf.

[84] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. "The SCADS director: scaling a distributed storage system under stringent performance requirements". In: *Proceedings of the 9th USENIX conference on File and stroage technologies*. FAST'11. San Jose, California: USENIX Association, 2011, pp. 12–12. ISBN: 978-1-931971-82-9. URL: http://dl.acm.org/citation.cfm?id=1960475.1960487.

[85] Patrick Valduriez. "Join indices". In: *ACM Trans. Database Syst.* 12.2 (June 1987), pp. 218–246. ISSN: 0362-5915. DOI: 10.1145/22952.22955. URL: http://doi.acm.org/10.1145/22952.22955.

[86] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective". In: *CIDR*. www.cidrdb.org, 2011, pp. 134–143.

[87] Dean Wampler and Alex Payne. *Programming Scala: Scalability= Functional Programming+ Objects*. O'Reilly Media, 2009.

[88] Kevin Weil. *Measuring Tweets*. http://blog.twitter.com/2010/02/measuring-tweets.html.

[89] Adam Wiggins. *SQL Databases Don't Scale*. URL: `http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/`.

[90] Jian Yang, Kamalakar Karlapalem, and Qing Li. "Algorithms for Materialized View Design in Data Warehousing Environment". In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 136–145. ISBN: 1-55860-470-7. URL: `http://dl.acm.org/citation.cfm?id=645923.673657`.

Appendix A

Creative Commons License

This is the text of Creative Commons Attribution-NonCommercial-NoDerivs License, version 3.0

A.1 License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   (a) "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

(b) "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

(c) "Distribute" means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

(d) "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

(e) "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

(f) "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

(g) "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

(h) "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

(i) "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   (a) to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

   (b) to Distribute and Publicly Perform the Work including as incorporated in Collections.

   The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   (a) You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work.

You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.

(b) You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

(c) If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

(d) For the avoidance of doubt:

i Non-waivable Compulsory License Schemes. In those jurisdictions in which

the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii Voluntary License Schemes. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b).

(e) Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

   UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability.

   EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

(a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

(b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

(a) Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

(b) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

(c) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

(d) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

(e) The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the

applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.